Fragmentbasierte Softwarearchitekturen für Produktlinien

D i s s e r t a t i o n

zur Erlangung des Grades
Doctor rerum naturalium
(Dr. rer. nat.)

vorgelegt von
Marco Körner
aus Hameln

genehmigt vom Institut für Informatik
der Technischen Universität Clausthal

2017

Dissertation Clausthal, SSE-Dissertation 14, 2017

Vorsitzende(r) der Promotionskommission
Prof. Dr. Jörg Müller

Hauptberichterstatter/in
Prof. Dr. Andreas Rausch

2. Berichterstatter/in
Prof. Dr. Gregor Engels

Tag der mündlichen Prüfung: 16.05.2017

Für meine Eltern

Kurzfassung

In vielen Domänen gestaltet sich die Architektur-Entwicklung von komponentenbasierten Softwaresystemen zunehmend schwierig. Zum einen liegt das an der Zunahme der Komplexität der Systeme. Zum anderen muss oft nicht nur ein einzelnes Produkt sondern eine ganze Familie von Software-Produkten entwickelt werden. Beispielsweise wird in der Automobilindustrie Steuersoftware so entwickelt, dass sie in unterschiedlichen Fahrzeugmodellen eingesetzt werden kann.

Eine Entwicklung der Software als eine Softwareproduktlinie reduziert die Komplexität, indem das Potential für Wiederverwendung erhöht wird. In einer Softwareproduktlinie wird die gesamte Produkt-Familie betrachtet. Gemeinsamkeiten und Unterschiede werden identifiziert und in Variabilitätsmodellen festgehalten. Ein solches Variatbilitätsmodell ist das Feature-Modell. Ein Feature repräsentiert ein fachliches Merkmal, das ein Produkt der Softwareproduktlinie haben kann. Ein Feature-Modell fasst alle möglichen Kombinationen von Features in einem Modell zusammen.

Die Erstellung von Softwarearchitekturen für einzelne Produkte kann auch unter Berücksichtigung eines Feature-Modells weiterhin sehr schwierig sein. Bei komponentenbasierten Softwarearchitekturen wird durch die Verschaltung der Komponenten ein Beziehungsgeflecht erzeugt. Erfordert die Softwareproduktlinie ein hohes Maß an Variabilität in dem Beziehungsgeflecht, nimmt die Anzahl möglicher Komponentenverschaltungen stark zu. Ein Software-Architekt kann die Menge der Verschaltungen dann nicht mehr überblicken.

Das Ziel dieser Arbeit ist, durch Einführung eines modular aufgebauten Variabilitätsmodells für komponentenbasierte Softwarearchitekturen, die Arbeit des Software-Architekten zu vereinfachen. Einen wesentlichen Beitrag zur Lösung bilden Strukturfragmente. Strukturfragmente repräsentieren eine geeignete Verschaltung mehrerer Komponenten in einem bestimmten Kontext. Einerseits hilft ein Strukturfragment, die Menge der möglichen Komponentenverschaltungen einzuschränken. Andererseits vereinfacht es die Entwicklung der Softwareproduktlinie, da es ein wiederverwendbares Artefakt ist. Durch alternative Verschmelzungen von Strukturfragmenten werden Beziehungsgeflechte mehrerer Produkte defi-

niert. Zusammen mit Softwarekomponenten bilden die Strukturfragmente das Variabilitätsmodell für die Softwarearchitekturen.
Eine weitere Entlastung des Architekten wird durch eine Werkzeugunterstützung ermöglicht. Durch die Verbindung der Variabilitätsmodelle kann nach Vorgabe einer Menge von Features eine geeignete Softwarearchitektur automatisiert abgeleitet werden.
Die Beschreibungssprache für die Modellierung wird weitgehend durch ein in UML modelliertes Metamodell unter Zuhilfenahme von OCL-Ausdrücken spezifiziert.
In der Arbeit wird ein Werkzeugprototyp spezifiziert, der die Modellierung und Architekturableitung unterstützt. Die Machbarkeit des Konzeptes wird mit diesem Prototypen demonstriert.

Danksagung

Das Anfertigen einer Dissertation gehört naturgemäß nicht zu den einfachsten Aufgaben, die einem im Leben begegnen können. Das Gelingen dieser Arbeit liegt nicht zuletzt an einigen Personen, denen ich an dieser Stelle für Ihre Unterstützung danken möchte.
Allen voran sei hier Prof. Dr. Andreas Rausch genannt. Er ermöglichte die Teilnahme an interessanten Projekten, aus denen die Idee für diese Arbeit erwachsen ist. Ich bedanke mich für die großzügig eingeräumte Zeit zur Promotion sowie für die vielen Diskussionen, für die sich immer etwas Zeit fand. Bei Prof. Dr. Gregor Engels bedanke ich mich für die Übernahme des zweiten Gutachtens.
Ein großer Dank gebührt den Kollegen am IPSSE bzw. SSE. Die starke Gemeinschaft führte nicht nur zu einem regen Austausch wissenschaftlicher Erkenntnisse, sie sorgte ebenso für ein sehr angenehmes Arbeitsklima.
Für das Lesen der Vorabversionen dieser Dissertation und die vielen wertvollen Kommentare danke ich Dr. Falk Howar, Henrik Liesner und besonders herzlich Christian Wiegand, Dirk Herrling und Nicole Gründler.
Vielen Dank!

Inhaltsverzeichnis

Kapitel 1

Einführung

Heutzutage werden immer mehr Softwaresysteme so entwickelt, dass sie in unterschiedlichen Umgebungen eingesetzt werden können. Damit Software die dafür notwendige Flexibilität aufweist, müssen die entwickelten Artefakte ein hohes Maß an Variabilität erlauben. Die Bereitstellung dieser Variabilität betrifft viele Aspekte bei der Entwicklung und stellt ein komplexes Problem dar. Diese Arbeit betrachtet die Variabilitätsmodellierung und die Variantenbildung im Bereich der Softwarearchitektur einer feature-orientierten Softwareproduktlinie.

1.1 Motivation

Die Software, die im Bereich der eingebetteten Systeme zum Einsatz kommt, wird zunehmend komplexer. Ein Grund hierfür ist die kontinuierlich steigende Anzahl der Aufgaben, die diese Systeme erfüllen müssen [Wei15].
Die Automobilindustrie repräsentiert einen Bereich, in dem diese Entwicklung besonders gut zu beobachten ist. Seit in den 70er Jahren des 20. Jahrhunderts zum ersten Mal Software in einem Auto integriert wurde, nahm die Anzahl der durch Software realisierten Funktionen ständig zu, um Anforderungen aus verschiedenen Bereichen wie Sicherheit, Effizienz oder Komfort zu erfüllen. Die Anzahl der *Lines of Code* hat bei diesen Softwaresystemen die 100 Mio. Marke längst überschritten. Dazu kommt, dass die Funktionen immer stärker vernetzt sind und auf über 60 Mikrocontroller im Fahrzeug verteilt werden [Pre+07].
Alle bedeutenden Automobilhersteller haben mehr als nur ein Fahrzeugmodell im Produktportfolio. Es werden unterschiedliche Fahrzeugmodelle mit jeweils unterschiedlichen Ausstattungen angeboten. Das bedeutet, dass auch die Entwicklung der Softwaresysteme für die jeweiligen Fahrzeuge durch diesen Variantenreichtum erschwert wird. Zu der inhärenten „natürlichen Komplexität“ [Küb09] der einzelnen Systeme kommt noch ein

umfangreiches und komplexes Variantenmanagement hinzu. Durch den harten Konkurrenzkampf in der Automobilindustrie ist der Kosten- und Zeitdruck so hoch, dass eine komplette Neuentwicklung aller Systeme unmöglich ist.
Es liegt nahe, die bei der Hardwareentwicklung angewendete Methodik der Entwicklung als Produktlinie auch auf die Softwareentwicklung zu übertragen, um die Variantenvielfalt unter diesen Bedingungen beherrschbar zu machen. Einzelteile werden dafür so konstruiert, dass eine Wiederverwendung in verschiedenen Produkten möglich ist. Durch eine Kopplung dieser Einzelteile – zum Beispiel Softwarekomponenten – mit abstrakten Merkmalsbeschreibungen (sog. Features) ist es oft verhältnismäßig einfach zu bestimmen, welche Komponenten man für die Erstellung eines Produkts benötigt [Kan+90].
Bei sehr großen und variantenreichen Systemen ist die Entwicklung einer Produktarchitektur auf diese Weise dennoch sehr schwierig. Die Angabe einer Menge von Softwarekomponenten reicht bei Fahrzeugsoftware nicht aus, um eine spezielle Variante zu definieren. Einige Features lassen sich zum Beispiel nur durch spezielle Parametrierung einer Komponente realisieren. Die Komponente muss dafür über interne Variabilität verfügen. Andere Features betreffen einen größeren Ausschnitt des Softwaresystems. Sie werden nur realisiert, wenn eine bestimmte Menge von Softwarekomponenten auf spezielle Weise zueinander in Beziehung steht. Ein bestimmtes Feature wird in einem Produkt durch ein solches spezifisches Beziehungsgeflecht realisiert. In einem anderen Produkt kann dasselbe Feature durch ein Beziehungsgeflecht mit einer abweichenden Struktur realisiert werden. Die Systemstruktur der entsprechenden Produkte weicht daher voneinander ab. Soll die Beschreibung dieser Systemstruktur ebenfalls in mehreren Produkten wiederverwendet werden, muss sie ebenfalls über Variabilität verfügen.
Dabei kommt erschwerend hinzu, dass bei komplexeren Realisierungen von Features auch Überschneidungen existieren. Das bedeutet beispielsweise, dass dieselbe Komponente an der Realisierung verschiedener Features beteiligt sein kann. Eine eindeutige Zuordnung einer Komponente zu einem Feature ist somit unmöglich.
Die Komplexität bezüglich der Variabilität wird bei der Entwicklung von Softwaresystemen in der Automobildomäne in der Zukunft weiter zunehmen. Daher muss eine Möglichkeit gefunden werden, den Softwarearchitekten bei der Entwicklung einer Architektur im Kontext einer Softwareproduktlinie zu unterstützen. Insbesondere unter der Voraussetzung, die Entwicklungszeit und -kosten weiterhin senken zu müssen, wird die Komplexität ansonsten nicht mehr beherrschbar sein.

1.2 Zielsetzung

Das Kernziel dieser Dissertation ist die Beantwortung der folgenden Forschungsfrage:

Forschungsfrage
Wie kann man in einer feature-orientierten Softwareproduktlinie automatisiert komponentenbasierte Architekturen generieren?

Diese Arbeit betrachtet dabei das Problem der Modellierung von Variabilität in der Struktur einer Softwareproduktlinienarchitektur für eingebettete Systeme in der Automotive-Domäne, aus der nach Vorgabe einer Featuremenge eine konkrete Architekturvariante für ein Produkt abgeleitet werden kann. Die Softwareproduktlinienarchitektur umfasst alle möglichen Varianten von Softwareartefakten, aus denen die Produkte der Softwareproduktlinie aufgebaut sind. Die Variabilität der Struktur ist ein Teilaspekt der Architekturvariabilität. Die in dieser Arbeit berücksichtigten Strukturen beschreiben die im jeweiligen System eingesetzten Komponenten und deren Abhängigkeiten untereinander bezüglich der kommunizierten Informationen. Auch aus anderen Perspektiven ist Variabilität in der Softwareproduktlinienarchitektur notwendig. Diese Variabilität, die man zum Beispiel in Verhaltensbeschreibungen oder Artefakten wie Anforderungen oder Testfallspezifikationen beobachten kann, liegt nicht im Fokus dieser Dissertation.
In der Automotive-Domäne spielt die Sicht auf die beschriebene Struktur eine besonders wichtige Rolle. Die eingebetteten Systeme realisieren dort in der Regel steuerungs- und regelungstechnische Funktionen. Im Bereich der Regelungstechnik werden Systeme üblicherweise anhand von Blockschaltbildern spezifiziert [Unb08]. Blockschaltbilder zeigen die Zerlegung in Subsysteme und den Signalfluss zwischen diesen. Sie dienen üblicherweise als Grundlage für die Strukturbeschreibung der Softwarearchitektur.
Die Variabilität hinsichtlich der Architekturbeschreibung wird in einem Softwareproduktlinienmodell repräsentiert. Aus diesem lassen sich sämtliche Produktarchitekturen ableiten. Um eine Ableitung automatisiert durchführen zu können, werden zusätzliche Informationen benötigt, die einen Bezug zu den Features herstellen. Damit kann durch eine Auswahl von Features die Variabilität im Modell aufgelöst werden.
Aus der Forschungsfrage ergeben sich vier Teilfragen, die in dieser Dissertation beantwortet werden. In welchen Formen tritt die Variabilität in der betrachteten Struktur der Softwareproduktlinienarchitektur auf? Das Ziel,

das mit dieser Frage anvisiert wird, ist eine allgemeingültige Klassifikation der Variabilität. Statt viele Einzelfälle zu betrachten, kann die Antwort auf die Forschungsfrage anhand der identifizierten Variabilitätsformen beschrieben werden. Die Antwort auf diese Frage ist ein wichtiger Schritt auf dem Weg zu einem generalisierten Modellierungskonzept.

Die zweite Frage lautet: Was sind die Ursachen für die Notwendigkeit unterschiedlicher Varianten von Architekturen in der Produktlinie? Verschiedene Produkte der Produktlinie, die unterschiedliche Features realisieren, führen zur Variabilität in einer Produktlinienarchitektur. Aber auch andere Aspekte, die nicht im Featuremodell abgebildet sind, können Variabilität verursachen. Der Fokus der Dissertation liegt auf dem Einfluss der Features auf die Variabilität in der Software. Aus diesem Grund werden die verschiedenen Ursachen strikt getrennt betrachtet.

Die dritte Frage zielt auf den Ablauf der Entwicklung einer Softwareproduktlinie. Wie ist ein Entwicklungsprozess aufgebaut, der sowohl die Modellierung der Softwareproduktlinie als auch die Ableitung einer Produktarchitektur umfasst? Der Prozess muss dabei sowohl die verschiedenen Variabilitätsformen wie auch die unterschiedlichen Ursachen für Variabilität berücksichtigen. Das bedeutet, dass bei der Erstellung des Softwareproduktlinienmodells für jede Variabilitätsform ein eigenständiger Prozessschritt vorgesehen ist. Bei der Ableitung einer Produktarchitektur ist für die Berücksichtigung verschiedener Variabilitätsursachen ebenfalls jeweils ein eigenständiger Prozessschritt vorgesehen.

Die vierte Frage adressiert die Modelle, die in diesem Prozess erstellt werden müssen. Wie kann ein Modell die Struktur einer Softwareproduktlinienarchitektur so beschreiben, dass eine konkrete Produktarchitektur daraus abgeleitet werden kann? Das Modell muss alle identifizierten Variabilitätsformen in der Struktur beschreiben können. Wichtig ist dabei, dass die verschiedenen Modellelemente für verschiedene Produktarchitekturen wiederverwendbar sind. Das bedeutet, dass nicht nur die Komponenten zu den wiederverwendbaren Softwareartefakten gehören. Auch die Beschreibungen der Beziehungsgeflechte, die mit den Komponenten die Struktur der Produktarchitekturen bilden, müssen in mehreren Produkten wiederverwendbar sein.

1.3 Aufbau der Arbeit

Im letzten Abschnitt dieses Kapitels wird kurz der Aufbau der Dissertation vorgestellt.

Das zweite Kapitel (*Grundlegende Konzepte*) vermittelt ein Grundwissen, welches für das Verständnis der Arbeit hilfreich ist. Dabei werden wichtige Konzepte und Begriffe aus den Bereichen *Softwarearchitektur*, *Softwareproduktlinien* und *Modellgetriebene Softwareentwicklung* erklärt und auf weiterführende Literatur verwiesen.
In Kapitel 3 (*Problemstellung und verwandte Arbeiten*) wird die Problemstellung der Arbeit näher erläutert. Das vorgestellte Beispiel einer Softwareproduktlinie für Automobil-Steuergeräte zeigt, wie stark sich die Architekturen verschiedener Produkte unterscheiden können. Das Beispiel ist zwar synthetisch und stark vereinfacht, es ist aber dennoch realistisch. Die verschiedenen Architekturen weisen die gleichen strukturellen Merkmale auf, die auch in realer Steuergerätesoftware existieren. Anhand dieses Beispiels werden drei verschiedene Variabilitätsformen und zwei Ursachen für Variabilität identifiziert. Ausschnitte aus diesem Beispiel werden durchgehend in dieser Dissertation für Erklärungen genutzt. In diesem Kapitel wird auch die Forschungsfrage aufgegriffen. Das damit definierte Kernziel der Arbeit wird durch die Definition von Teilzielen konkretisiert. Ein Überblick über verschiedene Lösungsmöglichkeiten zur Modellierung von Variabilität, die in anderen wissenschaftlichen Arbeiten beschrieben sind, bildet den Abschluss des Kapitels.
Das vierte Kapitel (*Der fragmentbasierte Lösungsansatz*) stellt das in dieser Arbeit entwickelte Modellierungskonzept vor. Anhand eines Entwicklungsprozesses wird demonstriert, in welcher Reihenfolge Artefakte modelliert werden müssen, um die Softwareproduktlinie zu beschreiben. Des Weiteren legt er die Schritte fest, mit denen aus dem Softwareproduktlinienmodell einzelne Produktarchitekturen abgeleitet werden können.
Die Kernidee für die Modellierung ist die Einführung von Strukturfragmenten. Strukturfragmente repräsentieren abstrakte Module zur Beschreibung von Beziehungsgeflechten in Architekturen. Mit der Bindung von Komponenten an Strukturfragmente werden Architekturbausteine hergestellt. Durch ein spezielles Verschmelzungsverfahren können die Architekturbausteine zu beliebigen Architekturen der Softwareproduktlinie zusammengesetzt werden.
Im fünften Kapitel (*Formalisierung des Konzeptes*) wird die Modellierungssprache formal vorgestellt, mit der sich eine fragmentbasierte Softwareproduktlinie beschreiben lässt. Dafür wird die abstrakte Syntax verschiedener Einzelteile durch ein UML-Metamodell spezifiziert. Zusätzliche Randbedingungen an das Modell werden durch OCL-Constraints festgelegt. Da die Modellierung der Softwareproduktlinie grafisch erfolgen soll, wird jeweils ein Vorschlag für eine konkrete Syntax in Form entsprechender Zeichnungen präsentiert.

Das darauf folgende sechste Kapitel (*Beschreibung eines Modellierungstools*) zeigt, wie ein Tool aussehen kann, das die entwickelte Modellierungssprache unterstützt. Es wird keine vollständige Spezifikation eines solchen Prototypen präsentiert, da diese den Rahmen der Arbeit sprengen würde. Es ist an dem vorgestellten Prototypen erkennbar, wie der Entwickler bei allen Schritten des Entwicklungsprozesses unterstützt werden kann.
Eine kleine Fallstudie im siebten Kapitel (*Fallstudie*) demonstriert die Anwendbarkeit des fragmentbasierten Modellierungskonzepts. Das Beispiel aus Kapitel 3 wird für diese Demonstration noch einmal aufgegriffen.
Schließlich werden im letzten Kapitel (*Zusammenfassung und Ausblick*) die Ergebnisse zusammengefasst und ein Ausblick auf mögliche weiterführende Forschungsarbeiten gegeben.

Kapitel 2

Grundlegende Konzepte

In dieser Arbeit wird ein Konzept vorgestellt, das bei der Entwicklung mehrerer großer Softwaresysteme einer Domäne unterstützt. Dafür wird auf verschiedene etablierte Konzepte und Begriffe zurückgegriffen. Für das Verständnis ist es hilfreich, wenn dem Leser die grundlegenden Konzepte *Softwarearchitektur*, *Softwareproduktlinie* oder *Variabilität* bekannt sind. Darum werden die wichtigsten Konzeptideen und Begriffe im Folgenden kompakt beschrieben.
Der erste Teil des Kapitels wird die Frage beantworten, was eine Softwarearchitektur ist. Das Thema „Softwareproduktlinie“ bildet den Fokus des zweiten Teils. Den Abschluss bildet ein Überblick über das Konzept „Modellbasierte Softwareentwicklung“.

2.1 Softwarearchitektur

Es gibt viele verschiedene Definitionen des Begriffs *Softwarearchitektur*. Sehr anschaulich zeigt das zum Beispiel die Definitionssammlung des *Software Engineering Institute* der Carnegie Mellon Universität in Pittsburgh, USA [16b]. Allein dort sind über 140 verschiedene Definitionen aus der *software engineering community* zusammengetragen worden. Dennoch lassen sich bei genauerer Betrachtung Gemeinsamkeiten erkennen.
Eine wichtige Aufgabe einer Softwarearchitektur ist die Beschreibung der grundlegenden Strukturen eines Softwaresystems. Die Definition von Bass, Clements und Kazman konzentriert sich auf diese Aufgabe.

Definition Softwarearchitektur (engl.: software architecture)
„The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationship among them.“ [BCK98]

Im Folgenden werden die einzelnen Begriffe dieser Definition beziehungsweise deren Bedeutung genauer betrachtet. Eine Software besteht aus Softwarekomponenten (*software components*). Komponenten kapseln einen Ausschnitt des Systems. Die darin enthaltene Funktionalität wird über wohldefinierte Schnittstellen nach außen bereitgestellt. Diese Eigenschaft macht eine Softwarekomponente zu einem in mehreren Produkten wiederverwendbaren Software-Teil, also zu einem „reusable asset“ [Szy98].
Um Komponenten korrekt einsetzen zu können, müssen einige Eigenschaften von außen sichtbar sein (*externally visible properties*). Zu diesen Eigenschaften gehören zum Beispiel eine Beschreibung der in der Komponente gekapselten Funktionalität oder die angebotenen Dienste [BCK98]. Wie diese Eigenschaften im Inneren umgesetzt werden, spielt bei der Betrachtung der jeweiligen Komponente in einer Architektur keine Rolle. Diese Information wird daher bewusst verborgen. Durch diese Form des „information hiding“ [Par72] realisiert eine Softwarekomponente eine Abstraktion. Eine komponentenbasierte Softwarearchitektur repräsentiert somit eine Abstraktion des gesamten Systems.
Komponenten interagieren in einem System ausschließlich über ihre Schnittstellen. In einem komponentenbasierten System spannen die Komponenten durch diese Interaktionen ein Beziehungsgeflecht auf (*relationships*). Das Beziehungsgeflecht ist in der Softwarearchitektur genauso wichtig wie die Menge der enthaltenen Komponenten. Die Eigenschaften von Systemen können sich verändern, wenn Komponenten auf unterschiedliche Art und Weise miteinander verbunden werden.
Die oben genannte Definition von Softwarearchitektur sagt aus, dass eine Softwarearchitektur aus mehreren Strukturen bestehen kann (*structure or structures*). Ein Beziehungsgeflecht, das zum Beispiel durch den Datenfluss zwischen verschiedenen Modulen definiert wird, bildet eine Struktur. Diese muss nicht die einzige relevante Struktur für die Architektur sein. Das Verhalten des Gesamtsystems zur Laufzeit lässt sich in Form von verschiedenen Prozessen beschreiben. Diese Prozesse müssen untereinander synchronisiert werden. Durch diese Synchronisation wird eine andere Art von Beziehung beschrieben, die daher eine andere Struktur repräsentiert. Beide Strukturen, weder die Modulstruktur noch die Prozessstruktur, beschreiben allein die Architektur [BCK98].
Obwohl die Softwarearchitektur-Definition von Bass, Clements und Kazman hinsichtlich der Strukturbeschreibung eines Systems sehr gelungen ist, greift sie etwas zu kurz. Die Softwarearchitektur hat einen maßgeblichen Einfluss auf die nichtfunktionalen Anforderungen an ein System wie zum Beispiel Wartbarkeit, Erweiterbarkeit oder auch Verständlichkeit. Sie beeinflusst somit auch Qualitätseigenschaften der Software [Gha+13].

Aus diesem Grund soll die Softwarearchitektur nicht nur als ein abstraktes Abbild eines Softwaresystems gesehen werden. Vielmehr legt sie fest, nach welchen Prinzipien eine Software entwickelt werden soll [SH09; Sta08; Her11].
Dieser Aspekt wird daher ebenfalls in Definitionen für Softwarearchitektur aufgenommen. Ein Beispiel hierfür ist die Definition des IEEE-Standard 1471-2000 (*IEEE Recommended Practice for Architectural Description for Software-Intensive Systems*). Die folgende Erweiterung dieser Definition wird von Mitgliedern des iSAQB[1] genutzt.

Definition Softwarearchitektur
„Die Softwarearchitektur definiert die grundlegenden Prinzipien und Regeln für die Organisation eines Systems sowie dessen Strukturierung in Bausteinen und Schnittstellen und deren Beziehungen zueinander wie auch zur Umgebung. Dadurch legt sie Richtlinien für den gesamten Systemlebenszyklus, angefangen bei der Analyse über Entwurf und Implementierung bis zu Betrieb und Weiterentwicklung, wie auch für die Entwicklungs- und Betriebsorganisation fest.“ [Gha+13]

Welche Strukturen für die Beschreibung des Systems relevant sind, wird unter anderem durch verschiedene Interessensvertreter (Stakeholder) bestimmt. Jeder Stakeholder betrachtet das System von einem anderen Standpunkt aus. Ein weitverbreitetes Konzept ist die Beschreibung eines Systems in verschiedenen Sichten auf die Architektur [Gha+13]. Eine Sicht repräsentiert dabei den Aufbau des Systems anhand ausgewählter Aspekte. Aspekte, die von einem Standpunkt aus gesehen nicht relevant sind, sind in der entsprechenden Sicht nicht dargestellt.
Es gibt verschiedene Modelle, die für die Darstellung eines Systems auf eine Menge von Standardsichten zurückgreifen. Das von Philippe Kruchten 1995 vorgestellte *4+1-Sichten-Softwarearchitekturmodell* beschreibt Systeme in fünf Sichten. Die Beschreibung in „4“ Hauptsichten (logische Sicht, Entwicklungssicht, Prozess-Sicht und Verteilungssicht) wird durch zusätzliche Szenarien unterstützt. Diese Szenarien bilden die fünfte Sicht („+1“), in der das Zusammenspiel der Systemteile anhand wichtiger Anwendungsfälle demonstriert wird [Kru95].
Ein verwandtes Sichtenmodell, das unter anderem im Arc42-Template oder im iSAQB-Lehrplan genutzt wird, beschreibt das System aus vier

[1] International Software Architecture Qualification Board (iSAQB) hat unter anderem das Ziel, die fachlich-inhaltliche „Qualität von Lehre, Aus- und Weiterbildung für Softwarearchitektur“ sicherzustellen (s. www.isaqb.org/association)

Sichten. Sehr oft bilden diese bereits ein gutes Fundament für eine Systembeschreibung. In der *Kontextsicht* wird das System in seinem fachlichen oder technischen Umfeld beschrieben. Die *Bausteinsicht* zeigt die statische Struktur des Systems, also die Zerlegung in mehrere Module (Bausteine) und deren Beziehung zueinander. Die Dynamik des Systems wird in der *Laufzeitsicht* dargestellt. Die Abläufe zwischen den Bausteinen sind in dieser Sicht von Interesse. In der *Verteilungssicht* wird die Abbildung von Softwareteilen auf die reale technische Infrastruktur präsentiert [HS12; Gha+13].

Für die Darstellung einer Sicht können grundsätzlich verschiedene Techniken eingesetzt werden. Es gibt sowohl textuelle als auch grafische Beschreibungssprachen [Gha+13; Sta08]. Als grafische Beschreibungssprache wird oft die Unified Modeling Language (UML) eingesetzt. UML hat den Vorteil, dass es diverse Diagrammtypen bietet. Für jede der vorgestellten Sichten existieren Diagrammtypen, die gut für die Darstellung der entsprechenden Sachverhalte geeignet sind. Für die Darstellung von statischen Strukturen eignen sich zum Beispiel Komponenten- oder Klassendiagramme, Sequenz- oder Aktivitätsdiagramme stellen besonders gut dynamische Aspekte dar, usw. [Obj12b; Obj12c].

2.2 Softwareproduktlinien

Das Ziel von Softwareproduktlinien ist die Reduktion von (Weiter-) Entwicklungskosten für eine Familie von Softwaresystemen einer Domäne [GBS01]. Grundsätzlich haben alle Systeme dieser Familie die gleiche Aufgabe zu erfüllen. Sie müssen dieses jedoch unter Berücksichtigung verschiedener Anforderungen durch Kunden oder durch Randbedingungen, die durch die Anwendungsumgebung gegeben sind, erfüllen. Das macht verschiedene System-Varianten erforderlich.

Die wesentliche Aufgabe bei der Entwicklung einer Softwareproduktlinie ist die Erfassung von Unterschieden und Gemeinsamkeiten der Systeme in dieser Familie. Das Wissen darüber kann bereits in einer frühen Entwicklungsphase berücksichtigt werden. Denn damit lassen sich Software-Artefakte erstellen, die insbesondere für die Wiederverwendung in unterschiedlichen Produkten derselben Familie geeignet sind.

Die folgende Definition für eine Softwareproduktlinie stammt von Linda Northrop vom *Software Engineering Institute* der *Carnegie Mellon University*.

Definition Softwareproduktlinie (engl.: software product line)
„A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“ [CN01]

Eine wichtige Aussage in dieser Definition ist, dass Features nur ganz spezielle Bedürfnisse an das System erfüllen (*features satisfy specific needs*). Aber was ist nun ein Feature? Die Definition von Kang et al. beschreibt es wie folgt:

Definition Feature
„A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.“ [Kan+90]

Features sind genau die Merkmale eines Softwaresystems, die für die betrachtete Domäne (*market segment or mission*) sichtbar und relevant sind. Features beschreiben ein System auf abstrakte Weise und betreffen immer fachliche Aspekte der Domäne. Technische Aspekte, die zum Beispiel nur für die Implementierung relevant sind, sind bei der Beschreibung eines Systems anhand von Features komplett unsichtbar.
Features werden realisiert durch eine gemeinsam nutzbare Menge an Artefakten (*common set of core assets*). Zu diesen Artefakten zählen zum Beispiel Softwarekomponenten und Architekturmodelle aber auch Dokumentation.
Da in einer Softwareproduktlinie die verschiedenen Produkte auf Basis einer gemeinsamen Menge (*common set*) von Features und *core assets* enwickelt werden, ist es notwendig, dass in beiden Bereichen Variabilität existiert. Der Variabilitätsbegriff beschreibt eine fundamentale Eigenschaft in einer Softwareproduktlinie. Aus diesem Grund darf eine Definition des Begriffs nicht fehlen. Die Folgende zeigt die wesentlichen Eigenschaften von Variabilität auf.

Definition Software Variabilität (engl.: software variability)
„Software variability is the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context.“ [SVB05]

Für das Management dieser Variabilität kommen Variabilitätssmodelle zum Einsatz [Sch+12]. Für die Variabilität auf fachlicher Ebene (die sogenannte *problem space variability*), die ihre Ursache in verschiedenen Anforderungen der Domänen-Experten hat, werden in der Regel Featuremodelle oder Entscheidungsmodelle genutzt [Lyt+13]. Ein Featuremodell beschreibt die Produktlinie anhand abstrakter Merkmale, die ein System der Produktlinie haben kann. Entscheidungsmodelle bestehen aus einer Menge von Entscheidungen, die zu einem bestimmten Zeitpunkt getroffen werden müssen. Jede Entscheidung beinhaltet eine Menge von Antworten, aus denen ausgewählt wird [SRG11]. Sowohl ausgewählte Features als auch getroffene Entscheidungen beeinflussen das Design des jeweiligen Produkts. Sowohl ausgewählte Features als auch getroffene Entscheidungen beeinflussen das Design des jeweiligen Produkts.
Damit diese Beschreibung der fachlichen Variabilität effizient genutzt werden kann, muss beim Systemdesign eine entsprechende Flexibilität vorgesehen werden. Die Software-Artefakte, zum Beispiel Softwarekomponenten oder eine Architektur, aus denen die einzelnen Systeme erstellt werden können, verfügen ebenfalls über einen gewissen Grad an Variabilität (die sogenannte *solution space variability*). Komponenten und Architekturen können damit durch Anpassungen für mehrere Produkte in unterschiedlichen Umgebungen genutzt werden. Bei der Implementierung dieser Variabilität werden die in Abbildung 2.1 schematisch dargestellten Prinzipien genutzt.

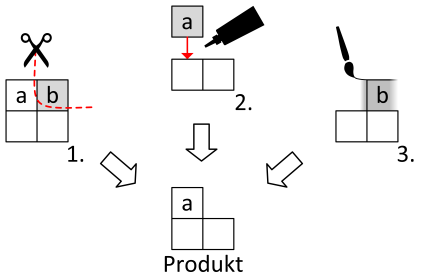


Abbildung 2.1: 1.) negative Variabilität, 2.) positive Variabilität und 3.) Variabilität durch Transformation

In [VG07] wird negative und positive Variabilität beschrieben. Bei *negativer Variabilität* existiert ein Modell, in dem alle möglichen Optionen enthalten sind. Durch Entfernen der nicht gewünschten Teile erhält man das Produkt. Im Vergleich dazu existiert bei *positiver Variabilität* nur ein gemeinsamer

Kern aller Produkte. Die gewünschten Optionen werden diesem hinzugefügt. Als dritte Möglichkeit, Variabilität zu erzeugen, wird zum Beispiel in [Hab+11a] Modelltransformation genutzt. Aus einem bestehenden Produkt wird durch Anwendung von Transformationsregeln ein anderes Produkt erzeugt. Andere Variabilitätsbeschreibungen (zum Beispiel aus [SD07; SVB05; Gal+11]) lassen sich auf diese drei zurückführen.
Ein Zusammenhang zwischen der Variabilität im *solution space* und der im *problem space* ist offensichtlich. Wenn diese Beziehung zwischen *solution space* und *problem space* explizit modelliert wird, kann die Erzeugung eines Produkts vereinfacht werden [Lyt+13].
Die Entwicklung von Produkten einer Produktlinie erfolgt in zwei Phasen. Die variablen Software-Artefakte (zum Beispiel eine Produktlinienarchitektur und Softwarekomponenten) sowie die Modelle zur Beschreibung der Variabilität auf abstrakterer Ebene beziehen sich auf Eigenschaften der gesamten Produktlinie. Die Erstellung dieser Modelle wird in der ersten Phase, dem sogenannten *domain engineering*, durchgeführt. Die Fertigstellung eines konkreten Systems der Produktlinie durch Auflösung der modellierten Variabilitäten (man sagt auch Instanziierung eines Produkts) wird in einem anderen Prozessschritt, dem sogenannten *application engineering*, durchgeführt. An dieser Stelle entscheidet man sich für eine konkrete Variante der Software.

2.3 Modellgetriebene Softwareentwicklung

Modelle bieten bei der Softwareentwicklung die Möglichkeit, auch komplexe Systeme vereinfacht darzustellen. Durch Abstraktion können die wesentlichen Aspekte eines Systems von einem ausgewählten Standpunkt aus isoliert in einem Modell beschrieben werden. Ein Modell, das beispielsweise nur die Struktur eines Softwaresystems beschreibt, sagt nichts über das Verhalten aus.
Das Ziel von modellgetriebener Softwareentwicklung ist die automatisierte Erzeugung von Software auf Basis von Modellen. Eine notwendige Bedingung hierfür ist die Beschreibung durch ein formales Modell, was in diesem Fall bedeutet, das es den jeweiligen Aspekt des Systems vollständig beschreibt [Sta+07]. Thomas Stahl, Markus Völter et al. definieren den Begriff wie folgt:

Definition Modellgetriebene Softwareentwicklung
„Modellgetriebene Softwareentwicklung (Model Driven Software Develop-

ment, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.“ [Sta+07]

Ein Beispiel für eine modellgetriebene Softwareentwicklung bildet der *Model Driven Architecture (MDA)*-Ansatz der Object Management Group (OMG)[2]. Den Kern dieses Ansatzes bilden verschiedene Modelle, die das zu entwickelnde Softwaresystem auf unterschiedlichen Abstraktionsebenen beschreiben. In Abbildung 2.2 sind die Zusammenhänge dieser Modelle dargestellt.
Das *Platform Independent Model (PIM)* beschreibt die Software idealerweise in einer domänenspezifischen Sprache. In dieser Abstraktionsebene werden technologische Aspekte (Programmiersprache, Middleware, etc.) nicht betrachtet. Durch eine Transformation in ein sogenanntes *Platform Specific Model (PSM)* wird das PIM durch zusätzliche Infomationen bezüglich der Technologie konkretisiert. Im letzten Schritt wird durch einen Codegenerator eine Transformation des PSM in Quellcode durchgeführt [Obj14; Sta08].

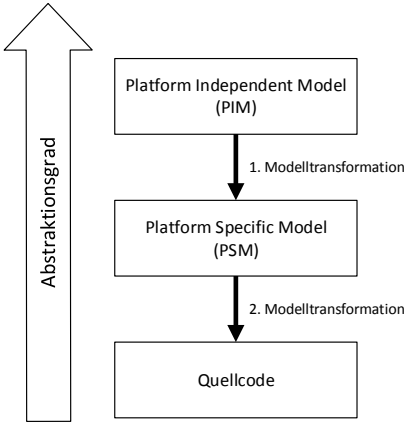


Abbildung 2.2: Die Modelle der verschiedenen Abstraktionsebenen bei MDA

[2]Der Begriff *Model Driven Architecture* darf nicht als automatische Architekturableitung verstanden werden [Sta08].

Ein wesentliches Konzept im MDA-Ansatz ist die Beschreibung eines sogenannten Metamodells. Das Metamodell definiert die Sprache, mit der das eigentliche Modell ausgedrückt wird. Es spezifiziert genauer gesagt die abstrakte Syntax der Modellierungssprache, also wie ein Modell aufgebaut sein muss. Ein Modell in MDA ist immer konform zu seinem Metamodell. Metamodellierung ist ein Grundkonzept der *Meta Object Facility (MOF)*, die ebenfalls von der OMG eingeführt wurde [OMG10; OMG15].
Die MOF besteht aus vier Meta-Ebenen, die jeweils beschreiben, wie Elemente der nächsttieferen Ebene aufgebaut sind. Auf der untersten Ebene (M0) befinden sich die konkreten Objekte. In der nächsthöheren Ebene (M1) wird die Struktur dieser Objekte beschrieben. Das bedeutet, dass die Elemente aus M0 eine Instanz von M1 sind. In Abbildung 2.3 ist dieser Zusammenhang als eine *instance of* Beziehung dargestellt. Die Strukturen des Modells aus M1 sind in der Ebene M2 durch ein Metamodell definiert. Das Metamodell spezifiziert die Modellierungssprache, in der das Modell beschrieben wird. Das Meta-Metamodell aus M3 beschreibt die Strukturen des Metamodells.

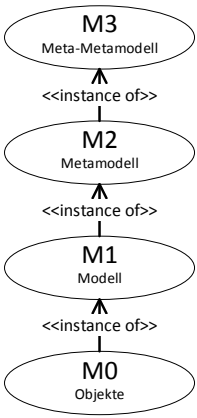


Abbildung 2.3: Aufbau der Meta Object Facility (MOF) - Jedes Element einer Ebene repräsentiert eine Instanz der Beschreibung der nächsthöheren Ebene

Die UML ist selbst nach diesem Prinzip spezifiziert [Obj12c]. Das bedeutet, sie ist als Modellierungssprache für den modellgetriebenen Modellierungsansatz grundsätzlich geeignet. Dennoch kann es sinnvoll sein, stattdessen eine

Sprache zu nutzen, die speziell an die Bedürfnisse einer Domäne angepasst ist. Domänenspezifische Sprachen (DSL) erleichtern die Beschreibung eines besonderen Aspekts eines Softwaresystems beziehungsweise des Systems selbst von einem speziellen Standpunkt aus [Kle08].
Eine abstrakte Syntax der DSL kann dabei zum Beispiel durch ein UML-Klassendiagramm spezifiziert werden. Durch die Anreicherung mit Ausdrücken in der Object Constraint Language (OCL) können zusätzliche Randbedingungen für das Modell spezifiziert werden [Obj12a; KW04].
Modellgetriebene Softwareentwicklung ist aufgrund der beschriebenen Möglichkeiten sehr gut für den Einsatz in der Entwicklung von Softwareproduktlinien geeignet [GS09].

2.4 Zusammenfassung

Softwarearchitektur ist mehr als nur ein Schnappschuss einer Software-Struktur. Die Softwarearchitektur beschreibt die Regeln und Prinzipien, nach denen ein Softwaresystem aufgebaut sein soll. Durch Benutzung unterschiedlicher Sichten können verschiedene Aspekte eines Systems wie das Verhalten oder die Struktur getrennt voneinander betrachtet werden. Jede Sicht kann grafisch oder textuell beschrieben werden.
Softwarearchitekturen einer Familie von Produkten einer Domäne weisen zum Teil zwar große Unterschiede auf, es gibt aber auch viele Gemeinsamkeiten. Softwareproduktlinien greifen diese Eigenschaften gezielt auf, um die Entwicklung einzelner Produkte zu vereinfachen. Gemeinsame Eigenschaften werden durch wiederverwendbare Softwareartefakte realisiert. Diese müssen allerdings über ein gewisses Maß an Variabilität verfügen, damit sie in unterschiedlichen Umgebungen oder mit unterschiedlicher Funktionalität eingesetzt werden können. Die Realisierung der Variabilität basiert grundsätzlich auf den Konzepten positiver oder negativer Variabilität beziehungsweise auf Variabilität durch Transformation. Festgehalten wird Variabilität in geeigneten Variabilitätsmodellen.
Mit der modellgetriebenen Softwareentwicklung (MDSD) wird eine Technik vorgestellt, die es ermöglicht, aus Modellen automatisiert Softwaresysteme zu generieren. Der *Model Driven Architecture (MDA)*-Ansatz repräsentiert eine Möglichkeit einer MDSD. Der Ansatz unterstützt die Nutzung einer domänenspezifischen Sprache (DSL). Eine solche DSL kann durch Spezifikation eines Metamodells definiert werden.

Kapitel 3

Problemstellung und verwandte Arbeiten

Nachdem einige wichtige Grundlagen zu Softwarearchitekturen und den Eigenschaften einer Softwareproduktlinie vermittelt wurden, wird im ersten Teil des folgenden Abschnitts ein Beispiel einer Softwareproduktlinie vorgestellt. Anhand dieses Beispiels werden verschiedene Herausforderungen hinsichtlich der Variabilitätsmodellierung deutlich. Mit diesem Beispiel wird die Problemstellung der Arbeit hergeleitet.

Die präsentierte Produktlinie umfasst eingebettete Systeme der Automobilindustrie. Der Fokus liegt dabei auf Funktionen des Antriebsstrangs und der Abgasaufbereitung. Die verschiedenen Fahrzeugtypen mit ihren vielfältigen Ausstattungsmöglichkeiten bewirken, dass die Umgebungen dieser Softwaresysteme stark variieren. Gleichzeitig müssen die Softwaresysteme eine unterschiedliche Anzahl an Funktionen bereitstellen. Das macht ein hohes Maß an Variabilität in der Software erforderlich. Die entwickelte Softwareproduktlinie stellt die dafür notwendige Flexibilität bereit.

Das Beispiel einer Steuergerätesoftware ist angelehnt an ein Projekt, das in Zusammenarbeit mit der Volkswagen AG bearbeitet wurde. Für die Nutzung in dieser Arbeit wurde es stark vereinfacht. Trotz seiner Einfachheit lassen sich verschiedene Herausforderungen bei der Beschreibung von Softwarearchitekturen im Kontext einer Softwareproduktlinie beobachten.

Im zweiten Teil des Kapitels werden Beobachtungen, die für die vorgestellte Domäne von besonderer Bedeutung sind, generalisiert und klassifiziert. Unter Berücksichtigung dieser Ergebnisse werden konkrete Ziele, die in dieser Arbeit erreicht werden sollen, im dritten Teil definiert. Der letzte Teil des Kapitels beinhaltet eine Zusammenfassung verwandter Arbeiten. Hier wird beleuchtet, wie mit existierenden Ansätzen die Modellierung des *solution space* einer Softwareproduktlinie möglich ist.

3.1 Das begleitende Beispiel - Eine Fahrzeugfamilie

Wenn man sich heutzutage ein neues Auto kaufen will, kann man die Ausstattung des Fahrzeugs an die eigenen Wünsche anpassen. Die verschiedenen Ausstattungen umfassen dabei nicht nur verschiedene Formen und Farben. Vielmehr werden auch vermehrt Hardware- oder Funktionsvarianten angeboten, die unterschiedliche Auswirkungen auf das Gesamtsoftwaresystem im Fahrzeug haben.

Im Beispiel kann ein Kunde auswählen, mit welchen Assistenz- oder Sicherheitssystemen das Auto ausgestattet wird. Diese Systeme nutzen im Betrieb die gleichen physikalischen Komponenten wie der Fahrer, so zum Beispiel die Bremsanlage oder den Motor. Damit das in jeder Situation sicher funktioniert, werden die verschiedenen Stellsignale durch Software koordiniert.

Des Weiteren kann der Käufer entscheiden, was für ein Antriebskonzept das Auto haben soll (Verbrennungsmotor und/oder Elektromotor) und ob er gegebenenfalls eine besondere Abgasaufbereitungsanlage wünscht. Auch diese Entscheidungen beeinflussen die Software. Die Umgebungen, in die die Softwaresysteme eingebettet sind, unterscheiden sich sehr deutlich. Dadurch unterscheiden sich auch die Anforderungen an die Software.

Die für die Produktlinie relevanten Entitäten, Eigenschaften und Zusammenhänge der Domäne aus Sicht des Domänenexperten definieren den *Kontext in der Domäne*. Daraus folgt, dass der Kontext verschiedene Eigenschaften der Softwareproduktlinie festlegt. Daher wird zunächst die allgemeine Funktionsweise der verschiedenen umgebenden Systeme und die Möglichkeiten der Interaktion mit dieser Umgebung erklärt.

Die Beschreibung der Softwareproduktlinie selbst besteht aus zwei Bereichen (s. Abbildung 3.1). Der *problem space* beschreibt die für den Kunden relevanten fachlichen Aspekte der Domäne in Form abstrakter Features. Anhand dieser Features werden die Gemeinsamkeiten und Unterschiede der einzelnen Produkte erfasst und in einem Featuremodell festgehalten.

Der zweite Bereich ist der *solution space*. Im *solution space* werden die tatsächlichen Softwareartefakte verwaltet. Diese bestehen aus der Beschreibung der verschiedenen Softwarearchitekturen und der darin genutzten Softwarekomponenten. Da die Anzahl der möglichen Architekturen im Softwareproduktlinien-Beispiel relativ groß ist, wird hier nur eine kleine Auswahl präsentiert. An diesen Exemplaren kann man erkennen, wie Features die Softwarearchitekturen beeinflussen. Am Ende des Abschnitts

wird die Bibliothek aller in den verschiedenen Architekturen genutzten Komponenten gezeigt.

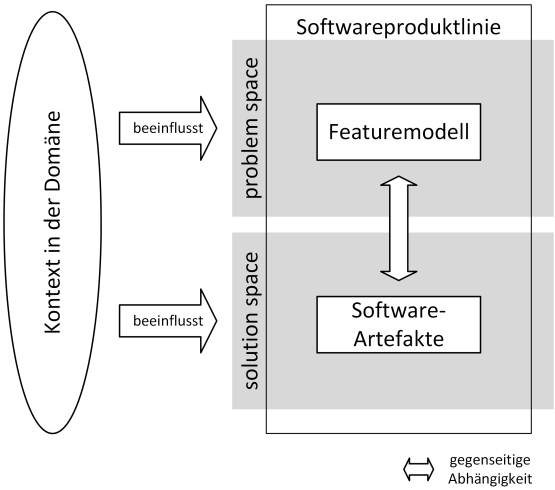


Abbildung 3.1: Abhängigkeiten zwischen *problem space* und *solution space*

3.1.1 Kontext in der Domäne

Die Softwaresysteme, die zu der vorgestellten Softwareproduktlinie gehören, fokussieren zwei Fachbereiche aus der Automobilentwicklung. Der erste ist die Längsdynamik, die ein Teilbereich der Fahrdynamik ist. In diesem Teil befasst man sich mit den Aspekten, die die Bewegung des Fahrzeugs in Längsrichtung betreffen. Dazu gehören sowohl Kräfte, die das Fahrzeug selbst durch den Antrieb oder die Bremsen erzeugt, als auch die Kräfte, die von außen auf das Fahrzeug wirken, wie zum Beispiel Hangabtriebskraft oder Strömungswiderstandskraft. Der zweite Bereich ist die Abgasaufbereitung. Diese hat zum Ziel, die Qualität der Emissionen des Verbrennungsmotors zu verbessern.

Längsdynamik

Für den Autofahrer ist die Einflussnahme auf die Längsdynamik sehr einfach. Möchte er die Geschwindigkeit erhöhen, tritt er stärker auf das Fahrpedal. Zur Reduzierung verringert der Fahrer entweder den Fahrpedalwinkel oder betätigt das Bremspedal.

Betrachtet man die physikalischen Vorgänge dabei, will der Fahrer eine Kraft erzeugen, die auf das Fahrzeug wirkt. Diese wirkt entweder in Fahrtrichtung, wenn er beschleunigen will, oder entgegengesetzt zum Verzögern. Aus fachlichen Gründen rechnet man im Bereich der Längsdynamik jedoch nicht mit *Kräften*, sondern mit *Momenten*. Der Fahrer stellt durch Betätigung der Pedale dementsprechend ein Moment ein, das das Fahrzeug aufbringen soll. Das Vorzeichen des Moments gibt die Wirkrichtung an. Ein positives Radmoment bewirkt eine Kraft, die nach vorn weist. Ein negatives Radmoment wirkt bei Vorwärtsfahrt entgegengesetzt der Drehrichtung der Räder.
Das Radmoment entsteht aus der Summe der auf das Fahrzeug wirkenden Kräfte. Ein positives Radmoment wird genau dann eingestellt, wenn der Antrieb eine Kraft in Fahrtrichtung bewirkt, die größer ist als alle entgegengesetzten Kräfte. Erzeugt der Antrieb das nicht, dreht sich die Wirkrichtung des Moments am Rad um und der entsprechende Wert wird negativ. Besonders stark ist dieser Effekt, wenn der Widerstand im Antrieb maximiert wird. Auf diese Weise bremst der Motor das Fahrzeug ab. Der Anteil des negativen Radmoments, der durch diese Motorbremse erzielt wird, ist das so genannte Schleppmoment.
Die Antriebe in den Beispielsystemen werden entweder durch Verbrennungsmotoren[1] oder durch Elektromotoren gestellt. Das größte Schleppmoment, das ein Verbrennungsmotor einstellen kann, wird durch eine Unterbrechung der Treibstoffzufuhr erreicht. Dann sind an der Antriebswelle nur noch die inneren Widerstände des Motors und anderer Bauteile feststellbar, die durch Reibung, Kompression der Luft in den Zylindern, etc. entstehen. Bei Elektromotoren sind noch viel größere Widerstände möglich, wenn sie in einen Generatormodus umschaltet sind. In diesem Fall wird die kinetische Energie des Fahrzeugs in elektrische umgewandelt. Der Vorteil neben der Erzeugung eines größeren negativen Moments ist die Möglichkeit, so einen Teil der zuvor investierten Energie zurückzugewinnen. Diese Energierückgewinnung beim Verzögern nennt man Rekuperation.
Den größten Anteil am negativen Radmoment kann man allerdings mit den Schleifbremsen erzeugen. Mit dem Bremspedal stellt der Fahrer den Druck in einem Hydrauliksystem ein. Die Größe des Moments, das von den Schleifbremsen aufgebracht wird, hängt von der Größe dieses Drucks ab.
Der Hydraulikdruck und somit auch das Bremsmoment kann dabei durch zusätzliche Systeme modifiziert werden. Zum einen kann ein elektronischer Bremskraftverstärker (EBKV) den Druck zusätzlich zum Fahrer erhöhen.

[1] Es handelt sich im Beispiel nur um Dieselmotoren.

Zum anderen kann die Hardware des elektronischen Stabilitätsprogramms (ESP) gezielt den Druck in den Zuleitungen zu den Bremsen der einzelnen Räder beeinflussen und somit auch das übertragene Moment an diesen Rädern verändern.
Für eine energieeffiziente Verzögerung wird versucht, einen großen Anteil des Verzögerungswunsches durch Rekuperation zu erzeugen. Der Rest wird durch die Schleifbremsen bewirkt. Die Verteilung des Moments an mehrere Aktoren erfordert jedoch, dass der Bremsdruck für die Schleifbremsen im Vergleich zu konventionellen, nicht rekuperierenden Systemen vermindert wird. Auch diese Druckminderung kann der EBKV einstellen.

Abgasaufbereitung

Gerade bei Dieselmotoren sind ohne Maßnahmen zur Abgasaufbereitung die Konzentrationen ungesunder Stoffe im Abgas wie beispielsweise Stickoxiden (NO_x) und Feinstaubpartikeln sehr hoch. Die Bemühungen der Fahrzeughersteller, die Emissionen von Schadstoffen ihrer Produkte zu minimieren, resultieren unter anderem in der Einführung effizienter und komplexer Katalysatorsysteme [Aut16]. Außerdem lassen sich Grenzwerte, deren Einhaltung durch die Gesetzgeber (zum Beispiel [Eur07]) gefordert wird, aktuell nicht anders einhalten. Ein inzwischen von vielen Herstellern eingesetztes SCR-Verfahren (*Selective Catalytic Reduction*) reduziert die NO_x-Emissionen um bis zu 95% [Rob15]. Bei diesem Verfahren wird eine Harnstofflösung in das Abgassystem eingespritzt, das verschiedene Katalysatoren und Filter enthält.
Ein Anteil der Emissionen wird durch kleine Rußpartikel gebildet. Mit einem Dieselpartikelfilter (DPF) können diese aus dem Abgasstrom herausgefiltert werden. Der Filter besteht aus einer Keramik in einer speziellen Struktur, an deren Wänden sich die Rußpartikel anlagern. Dadurch steigt der Strömungswiderstand und der Abgasdruck vor dem Filter nimmt zu. Durch Messung dieses Drucks kann das Motorsteuergerät erfassen, wie viel Ruß abgelagert wurde. Damit der Filter nicht verstopft, leitet es Maßnahmen ein, um den Ruß zu ungefährlicherer Asche zu verbrennen. Dieser Verbrennungsprozess wird entweder durch eine Anhebung der Abgastemperatur zum Beispiel durch zusätzliche Kraftstoffeinspritzung gestartet oder durch die Beimischung eines Additivs in den Kraftstoff, das eine Reduktion der Abbrenntemperatur des Rußes bewirkt.
Um den Verbrennungsprozess zu optimieren, kann ein spezieller Oxidationskatalysator (OXI-Kat) vorgeschaltet werden, bei dem der Anteil am reaktionsfreudigeren NO_2 im Abgas durch Umwandlung aus Stickstoffmonoxid (NO) angehoben wird.

Ein SCR-System besteht aus der Harnstoff-Einspritzungsanlage und aus einem speziellen SCR-Katalysator. Die eingespritzte Harnstofflösung wird durch diesen Katalysator in Ammoniak und CO_2 umgewandelt. Das Ammoniak reagiert mit den im Abgas enthaltenen Stickoxiden (NO_x) zu Stickstoff und Wasser.

Die Montage der einzelnen Komponenten kann in unterschiedlicher Reihenfolge erfolgen. Beispielsweise kann die Düse zum Einspritzen der Harnstofflösung sowohl vor als auch hinter dem DPF angeordnet sein. Bei moderneren Fahrzeugen nutzt man beispielsweise bei einer Montage vor dem Filter die Turbulenzen, die im Filter entstehen, um die Effizienz bei der Reaktion zur Ammoniakerzeugung zu erhöhen.

Steuergeräte

Im betrachteten Ausschnitt der Domäne werden mehrere Steuergeräte eingesetzt, an denen die verschiedenen Sensoren und Aktoren angeschlossen sind. Der Grund hierfür ist unter anderem in der Entstehungsgeschichte der Systeme zu finden. Bevor die ersten Softwaresysteme integriert wurden, existierten im Bereich der Längsdynamik zwei unabhängige Moment-Übertragungssysteme, die so genannten Momentenpfade. Einer verband das Fahrpedal mit dem Motor, der andere das Bremspedal mit den Bremsen. Ursprünglich waren diese rein mechanisch realisiert.

Die Einführung erster Computersysteme erfolgte durch Einbau spezieller Steuergeräte in räumlicher Nähe zu den jeweiligen Sensoren und Aktoren. Die ersten Softwaresysteme konnten nur in einem Momentenpfad eingreifen, weshalb das Vorhandensein mehrerer Steuergeräte kein Problem darstellte (zum Beispiel griff die Funktion für das Antiblockiersystem nur im Bremsmomentenpfad ein, um ein Blockieren der Räder bei zu großem Bremsmomentwunsch des Fahrers zu verhindern). Diese Trennung der Hardwaresysteme besteht noch immer. Allerdings sind die Hardwarekomponenten inzwischen durch Feldbussysteme (zum Beispiel CAN oder FlexRay^{TM}) miteinander vernetzt. Die Verteilung des Softwaresystems auf mehrere Steuergeräte wird in diesem Beispiel berücksichtigt.

Das bedeutet für das Beispiel, dass ein Motorsteuergerät und ein Bremsensteuergerät vorgesehen sind. Am Motorsteuergerät sind neben den Sensor- und Aktoranschlüssen für die Motorsteuerung auch die zur Abgasaufbereitung, Umgebungserfassung und das Fahrpedal angebracht. Das Bremspedal ist zusammen mit Bewegungs- und Lagesensoren sowie Steuerungen für die Bremshydraulik am Bremsensteuergerät angeschlossen.

3.1.2 Problem space der Fahrzeugproduktlinie

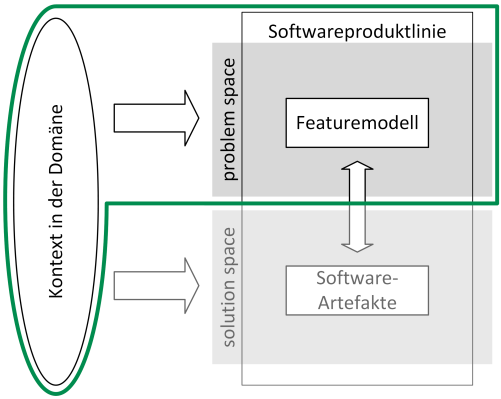


Abbildung 3.2: Im Fokus des Unterkapitels: Der Kontext der Domäne und das Featuremodell

Um den *problem space* der Softwareproduktlinie beschreiben zu können, werden zunächst die Funktionalitäten, die einzelne Systeme unter Berücksichtigung des Kontexts in der Domäne haben können, definiert (s. Abbildung 3.2). Die Gemeinsamkeiten und Unterschiede der verschiedenen Systeme hinsichtlich dieser Funktionalitäten eignen sich zur Identifikation einzelner Features. Im Folgenden werden die Funktionen beschrieben und das Featuremodell der Domäne vorgestellt.

Beschreibung der Funktionalitäten

Die Funktionalitäten, die von den verschiedenen Softwaresystemen in der Produktlinie bereitgestellt werden, lassen sich in die vier Teilbereiche *Komfort*, *Sicherheit*, *Antrieb* und *Umweltschutz* unterteilen.
Komfortfunktionen übernehmen auf Wunsch die Kontrolle über die Einstellung einzelner Fahrzeugparameter wie zum Beispiel der Geschwindigkeit. Dabei übernehmen sie aber nur Aufgaben, die der Fahrer grundsätzlich auch selbst erledigen kann.
Als Komfortfunktionen ist eine Geschwindigkeitsregelanlage (GRA) und ein *adaptive-cruise-control*-System (ACC) vorgesehen. Ein GRA kann ohne Eingriffe durch den Fahrer das Fahrzeug auf eine zuvor eingestellte Geschwindigkeit beschleunigen und diese halten. Es fordert wie der Fahrer vom Motor ein entsprechendes Moment an. Das ACC kann ebenfalls eine

Fahrzeuggeschwindigkeit einregeln, berücksichtigt dabei allerdings die Geschwindigkeit vorausfahrender Fahrzeuge. Um immer einen ausreichenden Sicherheitsabstand einzuhalten, passt das ACC die Sollgeschwindigkeit eigenständig an die Verkehrssituation an.

Der Fahrkomfort steigt dadurch, weil der Fahrer nur kontrollieren muss, ob die Situation in der Umgebung die Einstellung des Komfortsystems erlaubt. Der Fahrer kann diese Funktionen jederzeit unterbrechen, um selbst die Kontrolle zu übernehmen.

Sicherheitsfunktionen können ebenfalls einzelne Fahrzeugarameter einstellen. Im Gegensatz zu Komfortfunktionen lassen sich diese Funktionen aber nicht manuell aktivieren oder deaktivieren. Sie greifen automatisch ein, wenn eine unmittelbare Gefahr droht, die der Fahrer alleine prinzipiell nicht abwenden kann. Hierzu zählt zum Beispiel der Verlust der Bodenhaftung durch blockierende Räder.

Ein Antiblockiersystem (ABS) kann das einzustellende Bremsmoment begrenzen. Dadurch wird erreicht, dass der negative Schlupf der Räder nicht so stark wird, dass sie beim Bremsen blockieren. Da auch die Antriebsaggregate einen ähnlich kritischen Schlupf verursachen können, greifen weitere Schlupfregler durch Veränderung der angeforderten Motormomente rechtzeitig ein, sodass die Räder die Bodenhaftung nicht verlieren. Beide Sicherheitsfunktionen greifen direkt in Längsdynamiksysteme ein.

Ein indirekter Eingriff kann durch einen Gierratenregler erfolgen. Dieser hat eigentlich die Aufgabe, die Drehung des Fahrzeugs um die Hochachse zu regeln[2], stellt diese Drehung jedoch durch gezieltes Bremsen einzelner Räder ein, was sich auf die Geschwindigkeit des Fahrzeugs auswirkt.

Im Beispiel stehen mit Verbrennungsmotor und Elektromotor zwei verschiedene Antriebsarten zur Verfügung. Aus einer abstrakten technischen Sicht ähneln sich diese Teilsysteme sehr. Beide sind Hardwaresysteme der Umgebung, die sowohl ein positives wie auch negatives Moment erzeugen können.

Aus Sicht des Kunden ist die Wahl des Antriebs jedoch relevant. Verschiedene Aspekte bezüglich der Fahrzeugnutzung hängen stark vom verbauten Antriebssystem ab. Beispielsweise ist die Reichweite von Fahrzeugen mit Verbrennungsmotor aktuell deutlich höher als von E-Fahrzeugen. Die Anschaffungskosten sind beim Fahrzeug mit Elektromotor höher, dafür sind die Betriebskosten niedriger. Darum müssen die verschiedenen Antriebskonzepte als eigenständige Features der Produktlinie dem Kunden präsentiert werden.

[2] Das ist eine Funktion des Fahrdynamik-Bereichs *Querdynamik*.

Beim rekuperativen Bremsen wird die kinetische Energie des Fahrzeugs in elektrische Energie umgewandelt, die zum Laden des Akkus genutzt wird. Im Vergleich zu Schleifbremsen kann dieser Teil der Energie wiederverwendet werden, was einen positiven Effekt auf die Energiebilanz und somit indirekt auf die Umweltfreundlichkeit hat. Darum wird diese Funktion zu den Umweltschutzfunktionen gezählt. Die Hauptaufgabe der Software ist hierbei, das Moment in jeder Situation optimal auf die Aktoren zu verteilen. Es muss immer das gewünschte Moment erreicht werden, wobei das maximal Mögliche an Energie zurückgewonnen werden soll.

Die zweite Umweltschutzfunktion ist das SCR-System. Diese Funktion hat zwei Hauptaufgaben. Die erste ist die Dosierung der Harnstoffeinspritzung in den Abgasstrang, die zweite die Simulation der chemischen Prozesse im Abgas. Für die Berechnung der korrekten Dosierung benötigt die Software Informationen über die Zusammensetzung des Abgases an verschiedenen Stellen. Sensoren zur Messung dieser Werte haben aber zwei wesentliche Nachteile: Sie sind physikalisch bedingt nicht sehr genau und sie erhöhen die Anschaffungskosten für das Gesamtsystem. Da die chemischen Prozesse recht gut bekannt sind, kann man auf einige Sensoren verzichten. Unter Berücksichtigung des Motorzustands kann die Zusammensetzung des Abgases durch Simulation der chemischen Reaktionen hinreichend gut berechnet werden.

Featuremodell

Die Funktionen, die im vorherigen Abschnitt beschrieben sind, werden auf Features abgebildet. Anhand dieser kann der Kunde die einzelnen Produkte der SPL unterscheiden. Die verschiedenen Produkte realisieren unterschiedliche Mengen von Features. Eine solche Menge wird Feature-Konfiguration genannt. Die Unterschiede in Feature-Konfigurationen repräsentieren die Variabilität im *problem space*. Im Beispiel lässt sich diese Variabilität durch ein einfaches Featuremodell darstellen. Die Darstellung dieses Modells erfolgt durch einen Graphen, der eine Erweiterung der Darstellung aus der FODA-Studie darstellt [Kan+90]. Die einzelnen Features werden dabei monohierarchisch klassifiziert, weshalb die Grundstruktur einem Baum gleicht. Zusätzlich können Querbeziehungen zwischen beliebigen Features im Graphen hergestellt werden.

Es gibt verschiedene Arten von Kanten, die definieren, wie Feature-Konfigurationen mit den verbundenen Features aufgebaut sein können. Ein Feature (und gegebenenfalls der gesamte „Unterbaum“) ist entweder verpflichtend in jeder Feature-Konfiguration vorhanden oder optional.

Features können über miteinander verbundene Kanten einer Feature-Gruppe zugeordnet werden, die das Vorkommen der gruppierten Features in Feature-Konfigurationen genauer spezifizieren. Aus der Featuremenge einer *ALTERNATIV-Gruppe* muss genau ein Feature in jeder Feature-Konfiguration enthalten sein. Aus einer *ODER-Gruppe* können beliebig viele, mindestens aber eins der Features ausgewählt sein.

Querbeziehungen zwischen beliebigen Features werden durch gestrichelte Pfeile repräsentiert. Bei einer *requires*-Beziehung ist bei Auswahl des einen Features das entsprechend andere zwingend erforderlich. Ein gegenseitiger Ausschluss ist über eine *mutex*-Beziehung modellierbar.

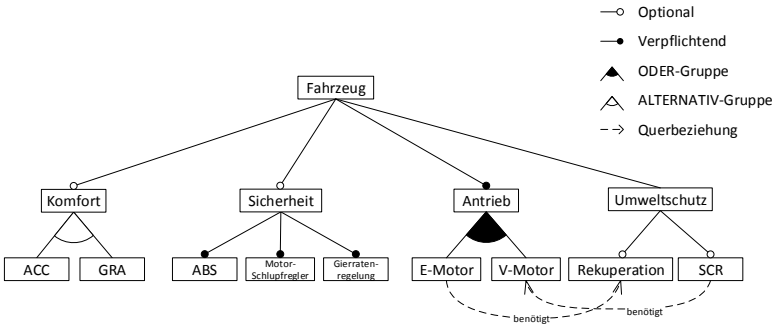


Abbildung 3.3: Featuremodell der Beispielproduktlinie

Aus den genannten Anforderungen und Randbedingungen des Beispiels lässt sich ein Featuremodell erstellen, das in Abbildung 3.3 dargestellt ist. Obwohl dieses Beispiel noch sehr übersichtlich ist, beschreibt es bereits 48 verschiedene Feature-Konfigurationen.

In der gewählten Baumstruktur werden alle Features unter dem Wurzelknoten *Fahrzeug* zusammengefasst. Darunter sind die vier Knoten *Komfort*, *Sicherheit*, *Antrieb* und *Umwelt* eingeführt, die die Unterteilung in unterschiedliche Fachdomänen beschreiben.

Die Features *ACC* und *GRA* repräsentieren Merkmale, die durch die Integration der entsprechenden Funktionen existieren. Sie sind in der Produktlinie beide optional, können aber nie gemeinsam in einem System vorhanden sein. Die Menge aller Sicherheitsfunktionen kann nur komplett im Softwaresystem vorhanden sein oder nicht. Das Feature *Sicherheit*

ist daher optional, und alle entsprechenden Unterfeatures (*ABS*, *Motorschlupfregler*, *Gierratenregelung*) sind als verpflichtend deklariert.
Mindestens einen Antrieb muss ein Auto haben. Es kann entweder ein Elektromotor, ein Verbrennungsmotor oder ein Hybridkonzept (*Elektromotor* **und** *Verbrennungsmotor*) gewählt werden. Die einzige Randbedingung dabei ist, dass bei Vorhandensein eines Elektromotors auch das Umweltfeature *Rekuperation* gewählt werden muss. Wenn kein Elektromotor gewählt ist, ist *Rekuperation* ein optionales Feature. Das zweite Umweltfeature *SCR* ist ebenfalls optional, allerdings ist für eine Auswahl notwendig, dass ein Verbrennungsmotor verbaut ist.
Darüber hinaus gibt es Anforderungen, die nicht explizit als Features im Featuremodell aufgenommen werden. Diese sind in jedem Produkt enthalten, und es kann zusätzlich vorausgesetzt werden, dass sie auch von den Kunden als Standard-Feature angesehen werden. Hierzu zählen zum Beispiel das Vorhandensein eines Fahrpedals beziehungsweise eines Bremspedals.

3.1.3 Solution space der Fahrzeugproduktlinie

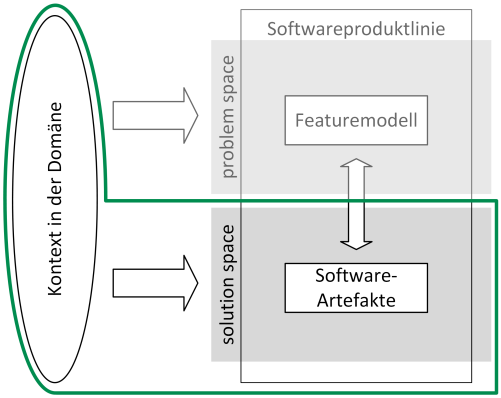


Abbildung 3.4: Im Fokus des Unterkapitels: Der Kontext der Domäne und die Software-Artefakte

Der *solution space* der Softwareproduktlinie wird aus den Software-Artefakten gebildet, die in den verschiedenen Produkten zum Einsatz kommen. Bei der Beschreibung der Artefakte wird auch der Kontext in der Domäne berücksichtigt (s. Abbildung 3.4). Da die Produkte des Beispiels kompo-

nentenbasiert entwickelt werden, sind Komponenten ein wesentlicher Teil dieser Artefakte.
Die Architekturen, die die Strukturen der einzelnen Produkte beschreiben, werden primär in einer Entwicklersicht präsentiert. Diese Sicht eignet sich zur Beschreibung des Beziehungsgeflechts zwischen Komponenten, das durch die notwendigen Interaktionen entsteht. In einer weiteren Sicht wird die Verteilung der Komponenten auf die verschiedenen Steuergeräte beschrieben.
Jedes Softwaresystem realisiert eine Feature-Konfiguration. Die große Anzahl an Feature-Konfigurationen führt dazu, dass es eine große Anzahl von Systemen gibt, deren Architektur unterschiedlich aufgebaut ist. Daher wird nicht das ganze Architekturmodell mit allen darin enthaltenen Softwarearchitekturen sondern nur eine kleine Auswahl Architekturen einzelner Systemen präsentiert. Diese Systemauswahl eignet sich gut, um verschiedene Aspekte der *solution space*-Variabilität zu zeigen.
Um beim Vergleich der Systeme die einzelnen Exemplare besser unterscheiden zu können, werden die drei Klassen „Luxus“, „Mittelklasse“ und „Billig“ eingeführt. Teurere Fahrzeuge bieten in der Regel mehr Features als günstigere. Das erste vorgestellte System bietet sehr viele Features, es ist ein Vertreter der Luxusklasse. Aus der Mittelklasse werden zwei verschiedene Ausstattungsvarianten präsentiert. Das einfachste System mit den wenigsten Features bildet das Modell „Billig“.

Beschreibung der Architekturen

Im Beispiel werden zwei Steuergeräte eingesetzt, die über einen Feldbus miteinander verbunden sein können: das Motorsteuergerät und das ESP-Steuergerät. Einige Komponenten bekommen die für ihre Aufgabe benötigten Daten direkt von Sensoren – andere steuern direkt Aktoren an. Die Anschlüsse der Sensoren und Aktoren sind fest den Steuergeräten zugeordnet. Darum ist es sinnvoll, die Komponenten den Steuergeräten zuzuordnen, an denen die benötigten Anschlüsse vorhanden sind.
Nur so können die Komponenten die teils harten Echtzeitanforderungen erfüllen. Allerdings bildet der Feldbus ein limitierendes Element (Flaschenhals). Die Menge der hierüber kommunizierten Daten muss begrenzt werden, was durch eine entsprechende Verteilung der Komponenten begünstigt werden muss.
Darüber hinaus werden die Komponenten voneinander getrennt, die unterschiedliche Arten von Features realisieren. Komponenten, die Sicherheitsfeatures realisieren, müssen strengere Qualitätsanforderungen erfüllen als andere. Durch eine räumliche Trennung wird die Sicherstellung der

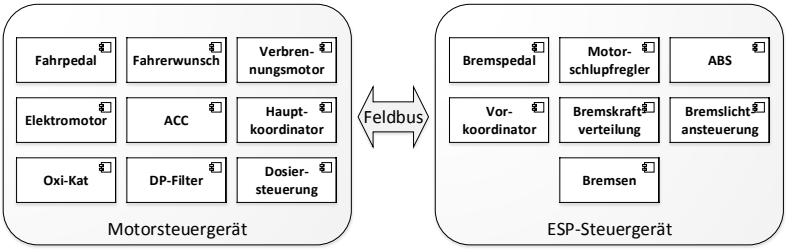


Abbildung 3.5: Verteilung der Komponenten eines Systems auf zwei Steuergeräte

sicherheitskritischen Anforderungen an die entsprechenden Komponenten vereinfacht (vgl. [Hil12] Seite 92 ff. [3]).

Abbildung 3.5 zeigt exemplarisch eine Komponentenverteilung einer Luxusvariante. Das System, das dort repräsentiert wird, realisiert eine bezüglich der Anzahl der Features größtmögliche Feature-Konfiguration. Sie enthält das Komfortfeature *ACC*, die Sicherheitsfeatures, beide Antriebsfeatures, also *Elektromotor* und *Verbrennungsmotor*, sowie die Umweltfeatures *Rekuperation* und *SCR*.

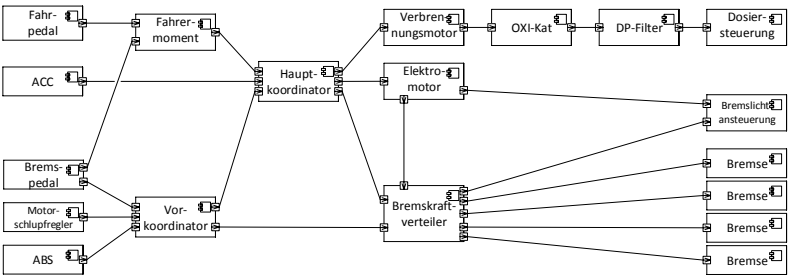


Abbildung 3.6: Komponentendiagramm der Luxus-Variante

[3] Diese Partitionierung der Software reduziert die gegenseitige Beeinflussung der Komponenten u.a. durch „Zuordnung von CPU Zeit, Speicherbereichen sowie Schnittstellen zur Kommunikation mit IOs“ [Hil12].

Es gibt Komponenten, die für die Realisierung eines Features benötigt werden. Diese haben den gleichen Namen wie das entsprechende Feature. Hierzu zählen die Komponenten *ACC*, *Motorschlupfregler* und *ABS*. Die Komponenten *Elektromotor* und *Verbrennungsmotor* sind an der Realisierung der entsprechenden Features beteiligt. Sie sind aber ebenso in der Lage, das Feature *Rekuperation* zu realisieren.
Das Feature *SCR* wird durch einen Verbund von Komponenten bereitgestellt. Dieser besteht aus *OXI-Kat*, *DP-Filter* und *Dosiersteuerung*. Die Komponenten *Fahrpedal*, *Bremspedal*, *Bremse* und *Bremslichtansteuerung* sind in jedem Fahrzeug vorhanden.
Das Besondere an der Komponente *Bremskraftverteilung* ist, dass sie zwar in jedem Fahrzeug enthalten ist, aber bei Auswahl des Features *Gierratenregelung* über eine erweiterte Funktion verfügen muss. Die Komponenten *Hauptkoordinator* und *Vorkoordinator* realisieren keine Features. Sie werden aus technischen Gründen benötigt.
Die Anordnung der Komponenten im Diagramm der Entwicklersicht (s. Abbildung 3.6) wird hauptsächlich durch die Wirkketten motiviert, also den Datenflüssen der regelungstechnisch wichtigen Größen, ausgehend von den Sensoren bis hin zu den Aktoren. Momentanforderer nutzen entsprechende Sensoren, weshalb sie auf der linken Seite zu finden sind, Aktoren steuernde Komponenten befinden sich rechts.

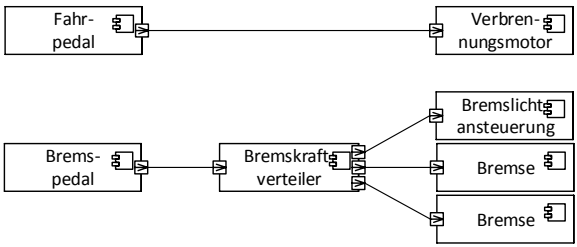


Abbildung 3.7: Die „Billig“-Variante benötigt keine Koordinatoren

Die Auswahl mehrerer Komfort- und Sicherheitsfeatures führt dazu, dass entsprechende momentanfordernde Komponenten enthalten sind. Eine wichtige Eigenschaft in diesem System ist, dass mehrere Anforderer gleichzeitig unterschiedliche Radmomente stellen wollen. Das ist physikalisch

nicht möglich. Berücksichtigt man diesen Umstand bei Entwurfszeit nicht, kann das zu indeterministischem Verhalten des Systems führen.

Die Koordinatoren stellen sicher, dass in jeder Situation nur eine Komponente Zugriff auf den entsprechenden Aktor bekommt. Sie legen dazu zur Laufzeit in Abhängigkeit von der jeweiligen Fahrsituation die Struktur der Wirkkette fest. Die Koordination der Momentanforderer wird durch die Komponenten *Vorkoordinator* und *Hauptkoordinator* realisiert. Die Verteilung des Radmoments auf die Aktoren erledigt das Komponentenpaar *Hauptkoordinator* und *Bremskraftverteilung*.

Für eine optimale Abgasaufbereitung durch SCR sind die Komponenten *Verbrennungsmotor*, *OXI-Kat*, *DP-Filter* und *Dosiersteuerung* in Reihe geschaltet. Die Reihenfolge dieser Anordnung entspricht der Reihenfolge, in der das Abgas die verschiedenen Hardware-Komponenten passiert.

Die Zusammensetzung des Abgases an einem Ort hängt immer von den physikalischen Komponenten ab, die das Abgas an dieser Stelle passiert hat. Dabei finden die Prozesse immer in einem kurzen Abschnitt des Abgasstrangs statt. Beispielsweise oxidiert das Sickstoffmonoxid immer beim Oxidationskatalysator zu NO_2.

Das erlaubt eine Dekomposition der Software in Komponenten, die jeweils für die Berechnung der Vorgänge in einem bestimmten Bereich des Abgasstrangs zuständig sind. Die Simulation des Reinigungsprozesses wird somit modular aufgebaut. Durch Hinzunahme der Einspritz-Steuerung lässt sich das Feature *SCR* auf diese Weise realisieren.

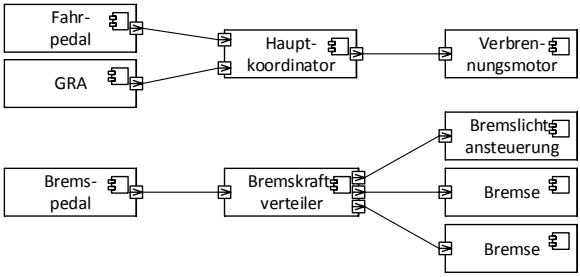


Abbildung 3.8: In dieser „Mittelklasse“ gibt es keine Interaktionen zwischen Antriebs- und Bremsmomentenpfad

Die Komponenten der Momentanforderer bzw. der Aktoren sind auf beide Steuergeräte verteilt. Da bei der Koordination und der Verteilung alle Anforderer und Aktoren berücksichtigt werden müssen, würden bei Nutzung einer einzelnen Koordinations- oder Verteilkomponente alle relevanten Daten über den Feldbus gesendet werden müssen. Dieser stellt im Gesamtsystem einen Flaschenhals dar, der durch eine verteilte Koordinationslösung deutlich entlastet werden kann. Es werden nur noch vorkoordinierte beziehungsweise vorverteilte Signale über den Bus gesendet. Auffällig sind bei einigen dieser Strukturen (zum Beispiel bei der Luxusklasse in Abbildung 3.6) redundante Wirkketten. Zum einen liefert die Komponente *Bremspedal* sowohl über *Fahrermoment* ihre Daten an den *Hauptkoordinator* als auch über den *Vorkoordinator*. Zum anderen existiert ein Pfad vom *Vorkoordinator* über den *Hauptkoordinator* zur *Bremskraftverteilung* als auch eine direkte Verbindung zwischen den beiden Komponenten. Diese Lösung ist Teil des Sicherheitskonzepts, das sicherstellt, dass bei Störung der Kommunikation über die Steuergerätegrenze hinweg ein alternativer Signalpfad sofort zur Verfügung steht. Die Koordinationskomponenten wählen jederzeit die bestmögliche Wirkkette aus.
Unter Berücksichtigung dieses Umschaltverhaltens ergibt sich ein weiterer Vorteil der Dezentralisierung der Koordinations- und Verteilungsfunktion. Auch bei Wegfall der Wirkketten, an denen jeweils Komponenten des anderen Steuergeräts beteiligt sind, existieren immer noch mehrere Möglichkeiten, die Aktoren zu steuern. Beispielsweise können *ABS* Komponente und der Fahrer über das *Bremspedal* verschiedene Momente fordern. Ein zusätzlicher Koordinator wäre auch bei einer zentralisierten Koordinationsfunktion erforderlich, dessen Funktion auch im Normalbetrieb redundant zu einer Teilfunktion des eigentlichen Koordinators gerechnet wird. Die verteilte Lösung verschiebt diese Teilfunktion explizit auf das ESP-Steuergerät, weshalb zwar die notwendigen redundanten Wirkketten erhalten bleiben, die unnötige Redundanz in der Berechnung aber vermieden wird.
Andere Systeme der Produktlinie weisen zum Teil erhebliche strukturelle Unterschiede auf. Zunächst sollen die Abweichungen von der soeben vorgestellten Architektur hinsichtlich der Momentkoordination präsentiert werden. Das erste alternative System beschreibt die einfachste Variante in der Produktlinie. Keins der optionalen Features wird durch dieses System realisiert, aus der Antriebsgruppe wird *Verbrennungsmotor* gewählt. Das entsprechende System repräsentiert ein „Billig“-Modell. Die Systemstruktur, die dafür benötigt wird, ist daher auch erwartungsgemäß sehr einfach, wie Abbildung 3.7 zeigt.

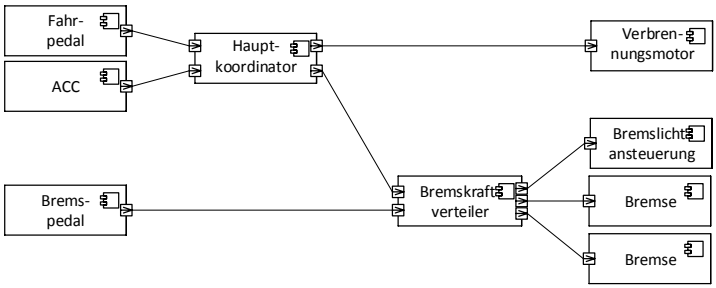


Abbildung 3.9: Das ACC dieser Mittelklasse fordert Kontrolle über zwei Aktoren

Das System zerfällt in dem Komponentendiagramm in zwei logisch unabhängige Teilsysteme. Da es keine zum Fahrer konkurrierenden momentanfordernde Funktionen gibt, ist eine Koordination nicht notwendig. Auch eine Verteilung einer Momentanforderung auf Antrieb und Bremsen ist in diesem System nicht sinnvoll, da die Verteilung des Moments bereits durch den Fahrer auf die physikalisch getrennten Momentenpfade geschieht.
Das in Abbildung 3.8 gezeigte Mittelklasse-System realisiert die Features *GRA* und *Verbrennungsmotor*. Das Komponentendiagramm zeigt, dass das System trotz mehrerer Momentanforderer noch immer aus zwei disjunkten Teilsystemen besteht. Da auf dem ESP-Steuergerät keine Sicherheitsfunktionen aktiv sind und auf dem Motorsteuergerät keine Funktionen, die die Bremsen nutzen wollen, ist eine Verbindung der beiden Systemteile nicht notwendig.
Vergleicht man diese Softwarearchitektur mit der des Systems, das statt des *GRA*-Features das *ACC* bietet, stellt man fest, dass, auch wenn es sich bei der Realisierung der Features jeweils um „einfache“ Komponenten handelt, die Struktur stärker abweicht als nur in der Benutzung einer anderen Komponente. Abbildung 3.9 zeigt, dass bei dem System mit einer *ACC*-Komponente eine zusätzliche Verbindung vom *Hauptkoordinator* zur *Bremskraftverteilung* existieren muss. Der Grund hierfür liegt in der Anforderung an das ACC, Bremseingriffe durchführen können zu müssen. Für die Auswahl der Funktionalität zur Abgasaufbereitung steht im Featuremodell genau ein optionales Feature zur Verfügung – das *SCR*-Feature. Bei der Realisierung müssen keine anderen Features berücksichtigt werden.

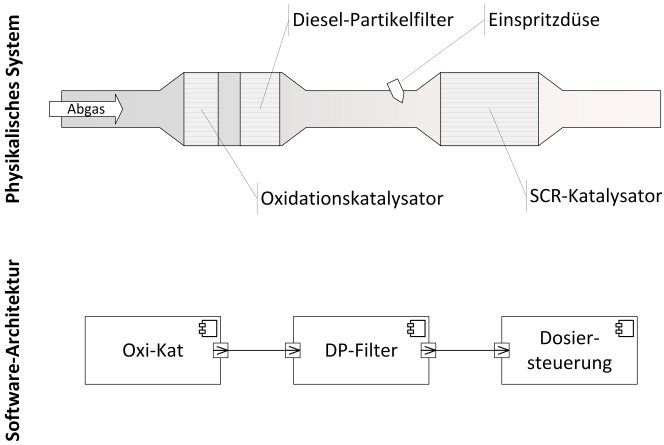


Abbildung 3.10: In dieser Variante befindet sich zwischen den Komponenten Oxi-Kat und der Dosiersteuerung die Partikelfilter-Komponente

Trotzdem gibt es unterschiedliche Softwarearchitekturen für die Realisierung dieses Features.

Beim Vergleich der Abbildungen 3.10 und 3.11 ist zu sehen, dass die entsprechenden Hardwarekomponenten in einer anderen Reihenfolge montiert sind. Das bedeutet, dass auch die Emulations- und Steuerungskomponenten in einer entsprechenden Reihenfolge angeordnet werden müssen. Wird zum Beispiel der Partikelfilter hinter der Einspritzeinheit montiert, darf auch die Berechnung des Filtereinflusses nicht vor der Dosiersteuerung erfolgen.

Die Modularisierung der SCR-Funktion ermöglicht also eine einfache Anpassung der Architektur an verschiedene Hardware-Systeme. Im Featuremodell ist dieser Unterschied nicht abgebildet; alle diese Systeme realisieren dasselbe Feature.

Komponenten-Bibliothek

Die Komponenten-Bibliothek enthält die Komponenten, aus denen die verschiedenen Systeme der Beispielproduktlinie aufgebaut werden können. Alle Softwaresysteme der entwickelten Produktlinie bestehen ausschließlich aus diesen Komponenten. In Tabelle 3.1 sind alle Komponenten aufgelis-

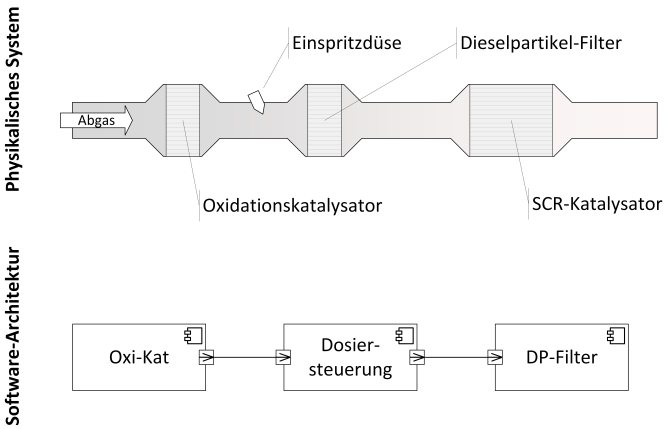


Abbildung 3.11: In dieser alternativen Variante folgt auf die Komponente des Oxi-Kat direkt die Dosiersteuerung

tet. Darüber hinaus wird dargestellt, welche Komponente direkt an der Realisierung eines Features beteiligt ist.
Beim Design der einzelnen Komponenten wurde eine hohe Kohäsion erreicht. Dadurch ist es möglich, einige Features durch eine eigenständige Komponente zu realisieren. Eine solche 1-zu-1-Beziehung existiert zwischen den Komponenten *ACC*, *GRA*, *ABS* sowie *Motorschlupfregler* und den jeweils gleichlautenden Features.
Die Komponenten Elektromotor und Verbrennungsmotor realisieren die Features *Elektromotor* und *Verbrennungsmotor*. Das Besondere an diesen Komponenten ist, dass beide ein alternatives Verhalten aufweisen können, das jeweils das *Rekuperation*s-Feature realisiert. Dieses Beispiel zeigt, dass eine n-zu-m-Beziehung zwischen Features und Komponenten existieren kann. Beispielsweise wird das Feature *Rekuperation* sowohl durch den Elektromotor als auch den Verbrennungsmotor realisiert (1-zu-m). Gleichzeitig kann der Elektromotor zwei Features realisieren (n-zu-1).
Auch die Komponente Bremskraftverteiler verfügt über alternative Verhaltensvarianten. Nur eine dieser Varianten realisiert das Feature *Gierratenregelung*. Für die Realisierung des *SCR*-Features müssen drei Komponenten im Verbund eingesetzt werden.
Alle anderen Komponenten realisieren keine identifizierten Features. Sie werden für die technische Realisierung der Systeme benötigt.

Nr.	Komponente	Features
1	ACC	ACC
2	GRA	GRA
3	Elektromotor	E-Motor / Rekuperation
4	Verbrennungsmotor	V-Motor / Rekuperation
5	ABS	ABS
6	Motorschlupfregler	Motorschlupfregler
7	Oxi-Kat	SCR
8	DP-Filter	SCR
9	Dosiersteuerung	SCR
10	Hauptkoordinator	-
11	Vorkoordinator	-
12	Bremskraftverteiler	Gierratenregelung
13	Bremspedal	-
14	Fahrpedal	-
15	Fahrerwunsch	-
16	Bremslichtansteuerung	-
17	Bremse	-

Tabelle 3.1: Komponenten der Softwareproduktlinie und die Features, die sie realisieren können

3.2 Variabilität im solution space

Der vorherige Abschnitt hat gezeigt, dass verschiedene Produktvarianten einer Softwareproduktlinie über grundverschiedene Softwarearchitekturen verfügen. Die Unterschiede zwischen diesen Architekturen definieren die notwendige Variabilität im *solution space* des Beispiels. Die Variabilität tritt in drei Formen auf: *Konfigurations-Variabilität*, *Komponenten-Variabilität* und *Systemstruktur-Variabilität*. Eine wesentliche Eigenschaft der betrachteten Variabilitätsformen ist, dass sie zur Entwicklungszeit aufgelöst werden müssen. Ähnliche Formen der Variabilität[4] werden auch bei dynamisch adaptiven Systemen unterschieden, wie zum Beispiel bei *DAiSI*-basierten Systemen [Nie+07]. Da die Variabilität bei Softwareproduktlinien im Gegensatz zur DAiSI nicht zur Laufzeit aufgelöst werden muss, bieten sich andere Modellierungsmöglichkeiten an.

[4] Im DAiSI-Konzept werden sie *Component Service Implementation Adaptation*, *Component Service Usage Adaptation* und *Component Configuration Adaptation* genannt [Nie+07].

Für die Notwendigkeit verschiedener Varianten gibt es verschiedene Ursachen (s. Abbildung 3.12). Die Auswahl von Features aus dem *problem space* bestimmt die *fachlichen* Aspekte, die bestimmte Varianten erforderlich machen. Der Kontext der Domäne liefert *technische* Aspekte, die als zusätzliche Randbedingungen für die Auswahl von Varianten berücksichtigt werden müssen.

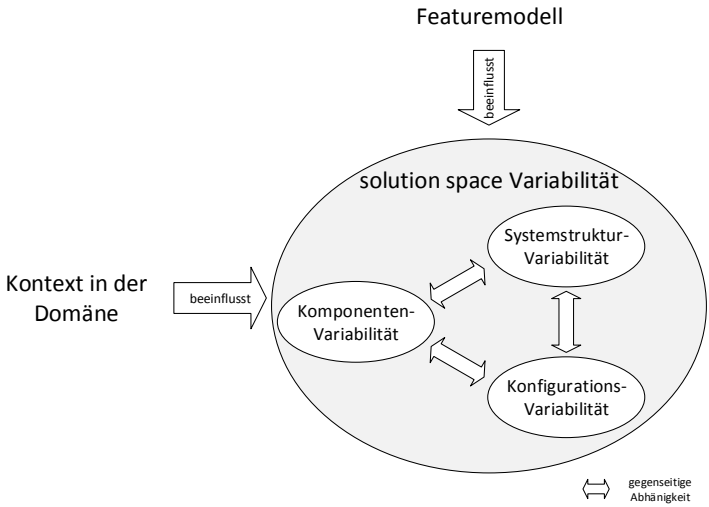


Abbildung 3.12: Formen und Ursachen der *solution space* Variabilität

3.2.1 Formen der solution space Variabilität

Diese drei Variabilitätsformen werden beim Vergleich von Ausschnitten der vorgestellten Softwarearchitekturen sichtbar. Die verschiedenen Formen adressieren jeweils einen anderen Ausschnitt der Softwarearchitektur. Die Konfigurations-Variabilität findet sich im „Inneren“ einer Komponente. Die Komponenten-Variabilität beschreibt die Variabilität in den Parts der Architektur. Die Systemstruktur-Variabilität wird in der Ebene der Beziehungsgeflechte sichtbar. Im Folgenden werden diese Variabilitätsformen mit Hilfe des Beispiels gezeigt.

Konfigurations-Variabilität

In Abbildung 3.13 werden zwei Ausschnitte von Softwarearchitekturen dargstellt. *Variante 1* zeigt die Hauptkoordinator-Komponente, wie sie im „Luxus“-System aus Abbildung 3.6 genutzt wird. Die zweite Variante zeigt die gleiche Komponente aus der zweiten vorgestellten Produktvariante „Mittelklasse“ (vgl. Abbildung 3.8). Ein offensichtlicher Unterschied zeigt sich in der Anzahl der mit der Umgebung verbundenen Ports.
Die Umgebungen, in die die Komponente jeweils eingebettet ist, unterscheiden sich. Damit der Hauptkoordinator in beiden Umgebungen eingesetzt werden kann, muss er über ausreichende Flexibilität verfügen. Diese notwendige Flexibilität wird dadurch erreicht, dass die Komponente über mehrere interne Konfigurationen verfügt.

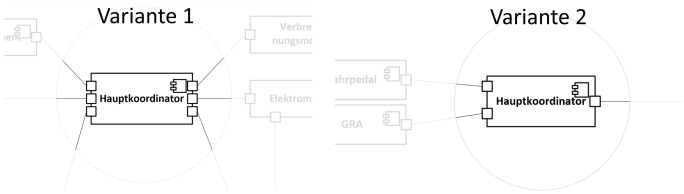


Abbildung 3.13: Konfigurations-Variabilität – Eine Komponente kann mehrere Konfigurationen haben

Damit kann zum einen die Funktionalität der Komponente über verschiedene Mengen von Ports erreichbar gemacht werden. Zum anderen können damit in einer Komponente alternative Funktionalitäten bereitgestellt werden.
In einer Softwareproduktlinie wird das ausgenutzt, indem das gewünschte beziehungsweise benötigte Verhalten mit der Feature-Konfiguration in Beziehung gesetzt wird. Diese Form der *solution space*-Variabilität wird im Folgenden *Konfigurations-Variabilität* genannt. Konfigurations-Variabilität beschreibt die unterschiedlichen Erscheinungsformen einer Komponente hinsichtlich ihrer „von außen“ erkennbaren Struktur (Ports) und ihres Verhaltens.

Komponenten-Variabilität

Die nächsten Architektur-Ausschnitte, die verglichen werden, stammen aus dem „Luxus“- und dem „Mittelklasse“-Modell (s. Abbildungen 3.6 und 3.8). Sie verfügen über unterschiedliche Komfortfeatures. In der Software-

architektur wirkt sich das auf die Auswahl der integrierten Komponenten aus. Beide benötigen zwar einen Koordinator, dieser koordiniert jedoch unterschiedliche Komponenten (s. Abbildung 3.14).
Das Systemverhalten kann durch Austausch einzelner kompatibler Komponenten verändert werden. Das bedeutet in diesem Fall, dass die ersetzende Komponente über eine Konfiguration verfügen muss, die über die gleiche Menge kompatibler Ports verfügt. In diesem Beispiel zeigt sich eine Stärke komponentenbasierter Architekturen.

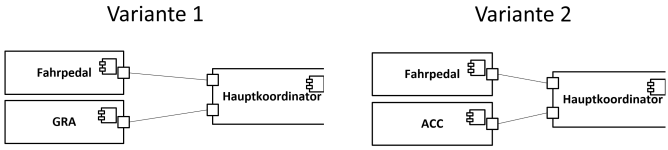


Abbildung 3.14: Komponentenvariabilität – An derselben Position können unterschiedliche kompatible Komponenten eingesetzt werden

Bei der Erstellung eines Softwaresystems der Produktlinie werden nur ganz bestimmte Komponenten benötigt. Durch die Verbindung der Komponenten mit dem Featuremodell kann die Menge benötigter Komponenten durch Auswahl einer Feature-Konfiguration festgelegt werden. Die Auswahl unterschiedlicher Feature-Konfigurationen führt zur Nutzung unterschiedlicher Komponentenmengen. Diese Form der *solution space*-Variabilität wird *Komponenten-Variabilität* genannt.

Systemstruktur-Variabilität

Zur Demonstration der dritten Variabilitätsform wird ein Vergleich der Luxusklasse mit einem Mittelklasse-Fahrzeug herangezogen (s. Abbildungen 3.6 und 3.8). Der Unterschied, auf den hier hingewiesen wird, besteht im unterschiedlichen Beziehungsgeflecht, das zwischen den Komponenten existiert (s. Abbildung 3.15). Auffällig ist, dass in beiden Varianten sowohl der *Hauptkoordinator* als auch die *Bremskraftverteilung* integriert sind. Nur in der ersten Variante existiert zwischen ihnen eine Kommunikations-Verbindung.
Der Einsatz von Komponenten in unterschiedlichen Verbindungsstrukturen hat ein abweichendes Systemverhalten zur Folge. Dabei kann die Komponentenmenge sogar gleich bleiben.

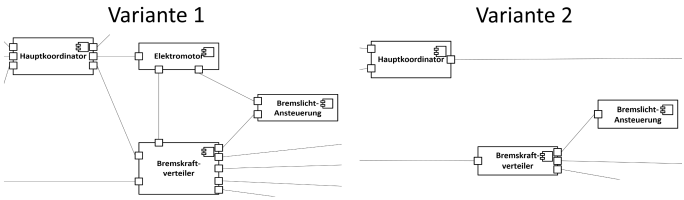


Abbildung 3.15: Systemstruktur-Variabilität – Die Komponenten eines Systems können unterschiedliche Beziehungsgeflechte bilden

Das Beispiel zeigt, dass ein Feature nicht nur durch eine bestimmte Auswahl von Komponenten und Konfigurationen realisiert wird. Auch die Form der Beziehungsgeflechte, in die die Komponenten eingebunden werden, bestimmt, welche Features realisiert werden. Es existiert also auch eine Verknüpfung verschiedener Beziehungsgeflechte mit dem Featuremodell. Eine Feature-Konfiguration legt dann fest, wie Komponenten miteinander in Beziehung stehen müssen.

Abhängigkeiten zwischen den Variabilitätsformen

In Abbildung 3.12 sind zwischen den verschiedenen Variabilitätsformen Abhängigkeiten dargestellt. Wie diese Abhängigkeiten entstehen können, ist in Abbildung 3.15 zu erkennen. Dort wird in den verschiedenen Varianten eine unterschiedliche Verbindungsstruktur benötigt. Diese lässt sich mit den ausgewählten Komponenten nur dann realisieren, wenn entsprechende Ports nutzbar sind. Der im Bild dargestellte Hauptkoordinator benötigt für die Interaktion mit dem Bremskraftverteiler einen zusätzlichen Port. Die zweite Variante in Abbildung 3.15 zeigt, dass nicht alle Konfigurationen des Koordinators über diesen Port verfügen.
Das bedeutet, dass in diesem Beispiel die Wahl der Verbindungsstruktur und die ausgewählte Komponente die Menge der nutzbaren Konfigurationen des Koordinators einschränkt.

3.2.2 Ursachen für verschiedene Varianten

Der Kontext der Domäne beschreibt unter anderem, wie die Umgebungen geformt sind, in die das jeweilige Softwaresystem eingebettet wird. Über die Variabilität in den Umgebungen kann der Kontext in der Domäne direkt das Featuremodell beeinflussen. Beispielsweise ist nur in einigen Hardwaresystemen ein Elektromotor verbaut. Diese Tatsache wirkt sich auf

das Featuremodell aus. Das Feature *E-Motor* wird als optionales Feature aufgenommen. Die Auswahl bestimmter Features wirkt sich dann auf die für die Implementierung notwendige Softwarearchitektur und somit auf die Variabilität des *solution space* aus.
Der Kontext beeinflusst die Variabilität im *solution space* aber auch direkt. Technische Randbedingungen, die nicht im Featuremodell aufgenommen werden, erfordern verschiedene Lösungen in der Softwarearchitektur.

Fachliche Aspekte des *problem space*

Features beschreiben unterschiedliche fachliche Aspekte der Domäne. Dass sich Softwarearchitekturen von verschiedenen Systemen unterscheiden, wenn diese unterschiedliche Features realisieren, ist offensichtlich.
Die Realisierung eines Features erfordert bestimmte Architekturteile, die bei Auswahl des entsprechenden Features in der Architektur vorhanden sein dürfen. Gibt es keine alternativen Implementierungen, müssen die Teile sogar vorhanden sein. Diese Architekturteile können dabei aus speziellen Komponenten, jeweils speziellen Konfigurationen von Komponenten oder speziellen Beziehungen zwischen Komponenten bestehen. Die notwendige Variabilität im *solution space* erstreckt sich also über alle vorgestellten Variabilitäts-Formen.
Durch das n-zu-m-mapping zwischen den Elementen des *solution space* und denen des *problem space* können Architekturteile mehrere Features realisieren.

Technische Aspekte des Kontextes in der Domäne

In einer Softwareproduktlinie kann es Unterschiede in den Architekturen geben, die nicht durch fachliche Aspekte – also durch unterschiedliche Feature-Auswahl – zu erklären sind. Der Grund hierfür sind technische Randbedingungen.
Zum einen betrifft das bestimmte Aspekte der Umgebung, die nicht als unterschiedliche Features im Featuremodell des *problem space* aufgenommen werden. Zum anderen kann eine bestimmte Kombination von Features in einer Feature-Konfiguration spezielle technische Lösungen erfordern. Im Beispiel wird das an mehreren Stellen deutlich.
Für die Realisierung des Features *SCR* beispielsweise gibt es verschiedene Lösungen. Die einzelnen Komponenten sind bezüglich der Anordnung im Beziehungsgeflecht vertauscht. Jede Variante ist aber nur für einen ganz bestimmten Aufbau des Abgassystems geeignet. Die unterschiedlichen

Hardware-Konstellationen werden im Featuremodell nicht berücksichtigt, sie müssen aber bei der Implementierung berücksichtigt werden.
Als Beispiel für die Lösung eines Feature-Interaction-Problems dienen die Koordinatoren. Bei einigen Feature-Konfigurationen müssen mehrere Funktionen integriert werden, die gleichzeitig und unabhängig voneinander dieselbe physikalische Größe ändern wollen. So können beispielsweise bei gleichzeitiger Integration der Features *ACC* und *ABS* die entsprechenden Komponenten zur Laufzeit unterschiedliche Radmomente anfordern. Das ist physikalisch nicht möglich, weshalb ein solcher Versuch nicht zugelassen werden darf. Die Lösung dieses Problems ist die Integration der Koordinatoren, die festlegen, welche Komponente zu welchem Zeitpunkt Zugriff auf die Aktoren bekommt.

3.3 Forschungsziele

Die Beobachtungen zeigen, dass bei der Entwicklung einer Softwareproduktlinie einige Herausforderungen gemeistert werden müssen. In dem nun folgenden Abschnitt wird erläutert, was die Dissertation zur Lösung der aus den Herausforderungen entstehenden Probleme beitragen soll. In dem ersten Abschnitt wird die Forschungsfrage genannt. Im Anschluss daran werden konkrete Ziele genannt, die in der Dissertation erreicht werden sollen.

3.3.1 Forschungsfrage

Das Beispiel zeigt, dass mit einem einfachen Featuremodell eine große Menge unterschiedlicher Systeme beschrieben werden kann. Die relativ einfache Beschreibung dieser Variabilität betrifft nur den *problem space*. Die Variabilität der Systeme im *solution space* kann weiterhin sehr komplex sein.
Ein komponentenbasierter Ansatz vereinfacht die Entwicklung solcher Systeme deutlich. Einzelne Software-Artefakte, die Softwarekomponenten, können in vielen Systemen der Softwareproduktlinie wiederverwendet werden. Bei großen Systemen aus variantenreichen Softwareproduktlinien ist der Aufwand zum Erstellen der Software auf Architekturebene trotzdem sehr hoch. Das ist insbesondere dann der Fall, wenn die Unterschiede der Architektur-Varianten alle vorgestellten Formen der *solution space*-Variabilität widerspiegeln.
Bereits kleine Unterschiede in der Menge der ausgewählten Features können deutliche Unterschiede in den entsprechenden Softwarearchitekturen

bewirken. Bei der Modellierung müssen dabei nicht nur die unterschiedlichen fachlichen sondern auch zusätzliche technische Randbedingungen berücksichtigt werden.
Obwohl die einzelnen Architekturen sich deutlich voneinander unterscheiden, sind die Teile, die ein bestimmtes Feature realisieren, durchaus ähnlich. Der Grund hierfür ist, dass innerhalb der Domäne ähnliche Aufgaben zu lösen sind. Die Lösungen sind sich daher ebenfalls ähnlich. Architekturmuster oder -stile bieten eine Möglichkeit, solche Teilstrukturen abstrakter zu beschreiben. Zusätzlich gibt es aber auch individuelle Lösungen die nur im Kontext der Produktlinie oder gar nur für einzelne Produkte Anwendung finden. Die Idee, Produkte aus wiederverwendbaren Modulen, die jeweils Teilstrukturen der Systeme beschreiben, zusammenzusetzen, liegt auf der Hand.
Der Aufwand bei der Architekturentwicklung ist aufgrund der großen Anzahl an Varianten trotzdem noch groß. Eine Unterstützung des Architekten durch ein geeignetes Werkzeug könnte diesen Aufwand deutlich reduzieren. Diese Annahme ist die Motivation für die folgende Forschungsfrage, deren Beantwortung das Kernziel dieser Dissertation ist.

Forschungsfrage
Wie kann man in einer feature-orientierten Softwareproduktlinie automatisiert komponentenbasierte Architekturen generieren?

3.3.2 Ziele der Arbeit

Zum Erreichen des Kernziels, die Beantwortung der Forschungsfrage, werden einige Teilziele definiert.
Die Entwicklung einer Softwareproduktlinie ist ein komplexes Unterfangen. Auch wenn ein Werkzeug dabei unterstützen kann, ist ein systematisches Vorgehen für den Erfolg unerlässlich. Darum muss als erstes ein solches Vorgehen beschrieben werden, in das das Werkzeug eingebettet wird.

Ziel 1
Es muss ein Entwicklungsprozess beschrieben werden, der die Schritte definiert, die für die Erstellung einer Softwareproduktlinie notwendig sind. Der Schwerpunkt liegt dabei auf der Architekturentwicklung.

Wichtig dabei ist, dass bewährte Konzepte der Softwareproduktlinien-Entwicklung aufgegriffen werden. Zu diesen Konzepten zählt zum Beispiel

die Auftrennung in die Hauptschritte *domain engineering* und *application engineering*. Auch die strikte Trennung der Beschreibung fachlicher Aspekte im *problem space* von der Beschreibung technischer Aspekte im *solution space* soll angewendet werden.
Zum einen erhöht der Einsatz bereits bewährter Konzepte für Teillösungen die Wahrscheinlichkeit für das Erreichen des Kernziels dieser Dissertation. Zum anderen ist zu erwarten, dass potentielle Anwender leichter Zugang zu der vorgestellten Lösung bekommen. Software-Entwickler, die bereits Erfahrungen mit der Entwicklung von Produktlinien haben, finden sich durch die gewohnten Strukturen schneller zurecht, da sie ihre Denkmuster nur in einzelnen Details anpassen müssen. Daraus kann eine höhere Akzeptanz bei der Einführung des Konzepts resultieren.

Ziel 2
Es soll ein formales Modellierungskonzept definiert werden, mit der sich die einzelnen Teile und Zusammenhänge modellieren lassen, sodass eine automatisierte Architekturableitung ermöglicht wird.

Die Modelle, aus denen Softwarearchitekturen mit speziellen Features abgeleitet werden können, müssen explizit modelliert werden. Hierfür wird eine Modellierungssprache benötigt. Für die Ableitung konkreter Architekturen muss des Weiteren ein Algorithmus bereitgestellt werden. Die Sprache und der Algorithmus werden in dieser Arbeit vorgestellt.
Eine wichtige Forderung ist, dass alle identifizierten Formen der Variabilität im *solution space* unterstützt werden. Durch die Möglichkeit, die verschiedenen Variabilitätsformen explizit modellieren zu können, kann die notwendige Flexibilität der Software-Teile für die Erstellung einzelner Produkte erreicht werden. Um die Wiederverwendbarkeit dieser Artefakte in der Produktlinie zu erhöhen, soll die Kopplung der Artefakte des *solution space* an die des *problem space* über n-zu-m-Mappings erfolgen.

Ziel 3
Anhand eines Werkzeugprototypen soll die Anwendbarkeit des Konzepts mit Hilfe des Beispiels demonstriert werden.

Das vorgestellte Beispiel wird genutzt, um zu demonstrieren, wie das Konzept in einer konkreten Softwareproduktlinie angewendet werden kann. Dazu wird ein Prototyp eines Werkzeugs spezifiziert. Dieser Prototyp erlaubt sowohl die Modellierung der einzelnen Modelle der Produktlinie wie auch die Ableitung konkreter Architekturen.

3.4 Verwandte Arbeiten

Es gibt bereits Werkzeuge, die zur Produktgenerierung sowohl im universitären als auch im industriellen Umfeld eingesetzt werden. Zu nennen sind hier unter anderem *Gears* [Inc15], *pure::variants* [pur15] und FeatureIDE [Thü+14]. Die Techniken zur Produktgenerierung, die in den Werkzeugen umgesetzt sind, basieren allerdings auf grundlegenden Konzepten, die in wissenschaftlichen Arbeiten behandelt sind. Im Folgenden werden daher die zugrunde liegenden Konzepte und Arbeiten beleuchtet.
Es gibt viele Arbeiten, die das Problem der Variabilitätsmodellierung unterschiedlich angehen. Im Folgenden werden die Wichtigsten kurz beschrieben. Hierbei wird der Schwerpunkt der Betrachtung auf die Modellierung der Variabilität im *solution space* gelegt. Die im Kapitel 2 vorgestellten Arten der Variabilitätsmodellierung „negative Variabilität“, „positive Variabilität“ und „Variabilität durch Transformation“ werden hier aufgegriffen, um die einzelnen Ansätze zu klassifizieren. Neben der Modellierung der Variabilität insbesondere auf Architekturebene wird die Möglichkeit der Verknüpfung der Variabilität im *solution space* mit der des *problem space* beleuchtet.
Ein wichtiges Ziel dieser Arbeit stellt die Ableitung einer Softwarearchitektur für ein konkretes Produkt der Produktlinie dar. Aus diesem Grund muss auch betrachtet werden, wie sich die Modellierungsansätze auf diesen Prozess, das *application engineering*, auswirken. Die Stärken und Schwächen der vorgestellten Arbeiten sollen bezüglich dieses Fokus identifiziert werden.

3.4.1 Negative Variabilität

Ansätze der *negativen Variabilität* basieren darauf, alle in der Produktlinie möglichen Implementierungs-Varianten gleichzeitig in einem Modell zu verwalten. Über Annotationen an den einzelnen Modell-Elementen wird festgelegt, bei welchen Feature-Konfigurationen das entsprechende Element im Produkt eingesetzt wird. Wenn beim *application engineering* eine Auswahl an Features getroffen wird, die das Produkt haben soll, werden die Elemente, deren annotierte Angaben nicht zur ausgewählten Feature-Konfiguration passen, vor der Instanziierung des Produkts aus dem Modell entfernt.
Auf diese Weise wird nicht nur die Existenz struktureller Elemente (Klassen, Komponenten, Assoziationen, ...) mit dem Featuremodell in Beziehung gesetzt. Auch Informationen wie Typ-Zuordnung oder Verhaltens-Alternativen können in einem Modell verwaltet werden. Weil alle Produkt-

varianten in einem Modell enthalten sind, spricht man bei diesem Ansatz auch von überlagerten Varianten (*superimposed variants*) oder auch von 150%-Modellen. Durch die Entfernung der für das zu entwickelnde Produkt nicht benötigten Modellelemente bei der Instanziierung realisiert dieser Ansatz eine Form der negativen Variabilität.
Ein Beispiel für diese Modellierungsweise der *solution space*-Variabilität ist der Ansatz von Michał Antkiewicz und Krzysztof Czarnecki [CA05]. Der *solution space* wird in Form eines so genannten Modell-Templates beschrieben. In Kombination mit einem Featuremodell[5] repräsentiert dieses Template eine gesamte Produkt-Familie.
Die Annotationen sind entweder durch boolesche Ausdrücke oder XPATH 2.0-Ausdrücke [Con10] realisiert. Damit lassen sich die Existenzbedingungen (*presence conditions*) beziehungsweise die Berechnung bestimmter Modellattribute (*meta-expressions*) in Abhängigkeit von Features und anderen Eigenschaften einer Featurekonfiguration (zum Beispiel die Anzahl von ausgewählten Features) beschreiben.

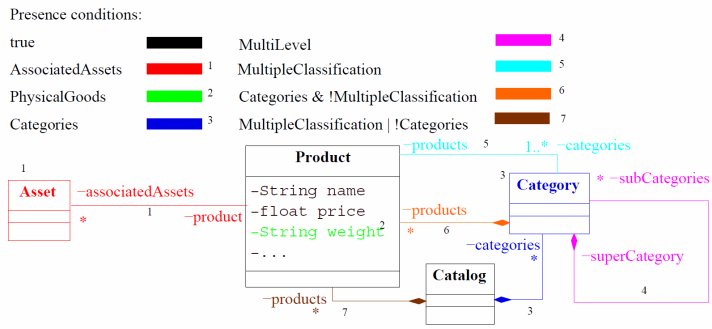


Abbildung 3.16: Beispiel eines Templates mit Annotationen (Bildquelle: [CA05])

In Abbildung 3.16 ist ein Ausschnitt eines Templates zu sehen, bei dem die annotierten Ausdrücke durch eine Farbcodierung den entsprechenden Modell-Elementen zugeordnet sind.
Ein weiteres Beispiel für Annotationen, das weit verbreitet Anwendung findet, ist der Einsatz von Präprozessor-Anweisungen wie zum Beispiel `#ifdef` in C-Code [Lie+10]. Über derartige Anweisungen im Quellcode

[5] In der genannten Arbeit wird ein *cardinality-based feature model* genutzt, das eine Erweiterung zur Featuremodellierung aus dem FODA Ansatz ist. Notwendig für die Art der Variabilitätsmodellierung im *solution space* ist das nicht.

lassen sich einzelne Befehle oder ganze Bereiche derart markieren, dass sie nur dann beim Kompilieren berücksichtigt werden, wenn die an die Anweisung geknüpfte Bedingung erfüllt ist. Diese Bedingung beschreibt eine Eigenschaft, welche die Feature-Konfiguration erfüllen muss, wie zum Beispiel die Existenz eines bestimmten Features.
Mit dem „Koala-Komponentenmodell“, das unter anderem in den „Philips Research Laboratories“ entwickelt wurde, ist es möglich, die Variabilität im Inneren einer Komponente zu modellieren [Omm+00]. Das Komponentenmodell ermöglicht eine hierarchische Strukturbeschreibung. Eine Komponente kann also aus mehreren Subkomponenten realisiert werden. Die Variabilität einer Komponente wird durch die Variabilität der Subkomponenten realisiert. Aus verschiedenen Varianten von Subkomponenten wird durch „Schalterelemente“ die erforderliche ausgewählt. Die Information, die für diese Auswahl benötigt wird, kann zwischen den Hierarchieebenen über *diversity interfaces* ausgetauscht werden. Die Erweiterung der Beschreibungssprache um diese speziellen Elemente kommt einer Annotation gleich.
Eine Generalisierung dieser Lösung wird durch Arne Haber vorgestellt [Hab+11c]. Auch dabei wird die Variabilität eines Systems durch die Variabilität der Komponenten, aus denen es besteht, beschrieben. Um die Variabilität einer Komponente zu modellieren, nutzt man bei der Komposition des „inneren Aufbaus“ entweder Variationspunkte oder man setzt variable Subkomponenten ein. Variationspunkte bestehen aus mehreren alternativen Varianten von Subkomponenten, aus denen nur eine beim *application engineering* gebunden wird. Bei variablen Subkomponenten wird die Variantenbindung an die nächste Hierarchieebene delegiert. Wie beim Koala-Ansatz wird auch hier die Variabilität in einer Hierarchieebene durch ein einziges Modell beschrieben. Um das Konzept anwenden zu können, wurde die Architektur-Beschreibungssprache Monticore um entsprechende Möglichkeiten erweitert.
Ein wesentlicher Vorteil eines annotativen Ansatzes auf Architekturebene ist die Flexibilität, die hinsichtlich der verschiedenen Formen der Strukturvariabilität geboten wird. Alle Variationsarten lassen sich grundsätzlich modellieren, und das, ohne dass der Entwickler eine weitere Sprache beherrschen muss.
Die Annotationen, über die eine Verknüpfung der Software-Artefakte mit den Features hergestellt wird, sind bei allen betrachteten Ansätzen so definierbar, dass ein *n-zu-m mapping* möglich ist.
Auf Nachteile von Annotationen wird am Beispiel des Präprozessor-Einsatzes schon seit den 90er Jahren immer wieder hingewiesen [SC92; AK09; LWE11]. Es kann vorkommen, dass die Implementierung eines Features

unter Umständen über den gesamten Code verstreut ist. Das Mapping zwischen *problem space* und *solution space* kann so schnell sehr komplex und daher schwierig zu beherrschen werden. Zusätzlich wird die Komplexität des eigentlichen Codes durch die Annotationen und die gleichzeitige Darstellung aller Varianten künstlich verschärft. Wenn man viele Varianten zu verwalten hat, die sich in allen Formen der Strukturvariabilität widerspiegeln, wird es sehr schwierig den Überblick zu behalten. Ein so erweiterter Code wird daher häufig mit „`#ifdef`-Hölle“ umschrieben [Loh+06].
Auch wenn der Abstraktionsgrad bei Annotationen im Code ein anderer ist als in den anderen genannten Ansätzen, lassen sich wesentliche Aspekte übertragen. Beispielsweise wird bei der Darstellung der Variabilität im Ansatz von Antkiewicz und Czarnecki im Modell ein zusätzliches Element in der konkreten Syntax verwendet: die Farbe. Die Variabilität ist ebensowenig modular modelliert und die künstliche Komplexität steigt bei zunehmendem Variantenreichtum deutlich an, weshalb die Annahme berechtigt ist, ein Skalierungsproblem bei dieser Lösung zu erwarten [Sch+12].
Bei der Modellierung von Variabilität, die nur für lokal beschränkte Feature-Implementierungsalternativen genutzt wird, ist der Einsatz von Annotationen zumindest bei einer geringen Anzahl an Varianten unter Umständen geeignet. Bei großen, variantenreichen Systemen ist diese Modellierungsart jedoch problematisch.

3.4.2 Positive Variabilität

Bei Ansätzen, die zur „positiven Variabilität“ zählen, wird während des *application engineering*s die Produktvariante aus verschiedenen Teilen zusammengesetzt. Zur Variabilitätsmodellierung kann diese Methode genutzt werden, indem man die einzelnen Systeme der Produktfamilie so dekomponiert, dass Softwareartefakte jeweils ein Feature realisieren. Im *application engineering* können anhand der Feature-Konfiguration nur die Teile selektiert werden, die für die Komposition des entsprechenden Produkts notwendig sind.
Der FORM-Ansatz (*Feature-Oriented Reuse Method*) von Kang [Kan+98] kann zu den kompositionalen Ansätzen gezählt werden. Er nutzt die Methodik aus FODA, um die Gemeinsamkeiten und Unterschiede der in der Softwareproduktlinie zusammenzufassenden Systeme auf Merkmalsebene zu identifizieren und in einem Featuremodell zu beschreiben.
Für die Entwicklung der Software-Artefakte, die diese Features realisieren sollen, wird als Richtlinie vorgegeben, dass die „physikalischen“ Grenzen in der Architektur aus den „logischen“ Grenzen, die durch das Feature-

modell definiert werden, folgen sollen. Dadurch kann man erreichen, dass die Variabilität im *solution space* modular modelliert wird. Unterschiedliche Features können so durch verschiedene wiederverwendbare Module realisiert werden, wobei die Wiederverwendbarkeit entweder durch fertige Komponenten, parametrierbare Templates oder Code-Skelette, die natürlich komplettiert werden müssen, realisiert wird.
Gemeinsamkeiten werden in einer so genannten Referenzarchitektur zusammengefasst. Diese Referenzarchitektur stellt ein für mehrere Systeme verwendbares Architekturtemplate dar. Durch Hinzufügen der Module, die produktspezifische Merkmale realisieren, wird die Softwarearchitektur eines Systems der Produktlinie erstellt. Es ist dabei möglich, mehrere Referenzarchitekturen für eine Produktlinie zu definieren.
Gelingt die Modularisierung des *solution space* unter Berücksichtigung der Features, ist ein Mapping zwischen den Software-Artefakten und den Features einfach. Es ist eine eins-zu-eins-Abbildung zwischen den Features und je einem Softwaremodul.
Für den Schritt des *application engineering*s folgt, dass bei Auswahl einer Feature-Konfiguration zum einen eine passende Referenzarchitektur ausgewählt wird und zum anderen eine Menge von Modulen, die „nur“ in der Referenzarchitektur an die entsprechende Stelle eingesteckt werden müssen. Die Prozesse der FORM-Methode sind in der Abbildung 3.17 dargestellt.

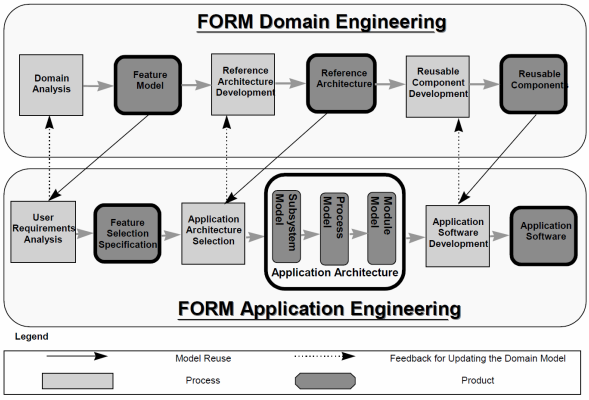


Abbildung 3.17: Die Entwicklungs-Prozesse der *Feature-Oriented Reuse Method* (Bildquelle: [Kan+98])

Julie Street Fant beschreibt in ihren Arbeiten, wie man Architektur- und Entwicklungsmuster nutzen kann, um die Variabilität zu modellieren

[FGP13; Fan11; SGG12]. Wie bei FORM werden bestimmte fachliche Merkmale durch Komponenten realisiert. Für die Beschreibung der gesamten Systemstruktur und des Systemverhaltens kommen Muster zum Einsatz, bei denen abstrakte Rollenbeschreibungen die Entitäten des zu erstellenden Softwaresystems repräsentieren. Durch die Information, welche Rollen eine Komponente erfüllen kann, lassen sich verschiedene Architekturen durch die Wahl unterschiedlicher Muster beschreiben.
Jede Komponente steht mit genau einem Feature in Beziehung. Um eine einfache und eindeutige Auswahl des passenden Musters im *application engineering* zu ermöglichen, erweitert Fant das Featuremodell um zusätzliche „nichtfunktionale Features“, die mit den Mustern verknüpft werden. So wird ein Mapping zwischen *problem space* und *solution space* durch eins-zu-eins-Abbildungen zwischen „funktionalen Features“ und Komponenten beziehungsweise „nichtfunktionalen Features“ und Mustern definiert.
Beim *application engineering* wird durch die Feature-Konfiguration festgelegt, nach welchem Muster die Software aufgebaut wird und welche Komponenten zum Einsatz kommen müssen. Über die Rolleninformation können Komponenten in einer Struktur komponiert werden, die durch das Muster vorgegeben ist.
Die vorgestellten Lösungen haben gemein, dass sie sehr gut geeignet sind, um die Variabilität auf Komponentenebene zu beschreiben. Beim *domain engineering* wird festgelegt, welches Feature durch welche Komponente realisiert wird. Die Bindung zwischen *problem space* und *solution space* ist dadurch trivial, weshalb dieses Konzept auch den *application engineering*-Schritt vereinfacht. Durch die eins-zu-eins-Abbildung wird allerdings die gewünschte Flexibilität nicht geboten.
Die Modellierung von Variabilität, die sich hauptsächlich auf das Beziehungsgeflecht auswirkt, ist grundsätzlich auch möglich. Bei FORM erstellt man dazu entsprechend mehrere Architekturtemplates, aus denen beim *application engineering* das passende gewählt wird. Ist der Anteil dieser Strukturvariabilitätsform jedoch hoch, benötigt man viele Architekturtemplates. Gerade bei großen und komplexen Systemen wird das den Modellierungsaufwand für die Architekturen im *domain engineering* anheben.
Mit der Idee, Muster für die Beschreibung von Verbindungsstrukturen zu nutzen, führt Julie Street Fant eine zusätzliche Abstraktionsebene ein. Das ist für die Beherrschung der Komplexität großer Systeme sehr hilfreich. Sie kann dadurch auch alle Formen der Strukturvariabilität darstellen. Allerdings nutzt auch sie eine eins-zu-eins-Abbildung zwischen Features und Komponenten, um die Variabilität des *problem space* mit der des *solution space* zu koppeln.

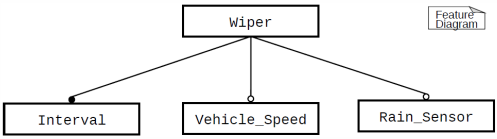
Zwar kann man auch die Variabilität auf struktureller Ebene modellieren, indem man entsprechend viele Muster nutzt, muss dafür aber entsprechend viele „nichtfunktionale Features“ im Featuremodell einführen. Diese nichtfunktionalen Features sind jedoch der wesentliche Mangel ihrer Lösung.
Muster bieten Lösungen auf einer technischen Ebene. Die Einführung nichtfunktionaler Features zur expliziten Beschreibung einer technischen Lösung im Featuremodell verletzt damit den Wunsch, im Featuremodell ausschließlich die Variabilität bezüglich fachlicher Aspekte zu verwalten. Dem Kunden wird damit bei der Erstellung einer Feature-Konfiguration im *application engineering*-Prozess nicht nur die Entscheidung überlassen, *was* das Produkt können soll, sondern er wird auch gezwungen, zum Teil zu entscheiden, *wie* es realisiert wird.

3.4.3 Variabilität durch Transformation (Delta-Modellierung)

Für die Erzeugung einer Produktvariante durch Transformation einer anderen Variante während des *application engineering*s werden beim *domain engineering* Änderungsvorgaben definiert. Die Menge der Vorgaben, die eine solche Transformation bewirken, definieren das Delta zwischen den beiden entsprechenden Varianten. Der Ausgangspunkt dieser Art der Variabilitätsmodellierung ist dabei in der Regel die Beschreibung eines konkreten Systems der Softwareproduktlinie[6].
Deltas, die Änderungsvorgaben zum Beispiel auf Architekturebene definieren, geben beispielsweise an, welche Komponenten und Beziehungen aus einem System entfernt oder hinzugefügt werden müssen, damit ein anderes System beschrieben wird. Die Deltas können an Elemente des Variabilitätsmodells im *problem space* gekoppelt werden. Angewendet werden die Änderungsvorgaben beim *application engineering*. Durch die Verknüpfung mit Features wird mit der Feature-Konfiguration die entsprechende Deltamenge selektiert. Grundsätzlich ist es möglich, die Anwendung der Deltas zu kaskadieren, sodass ein bereits transformiertes Modell als Basis für weitere Transformationen dienen kann. Daraus entstehen Abhängigkeiten zwischen den Deltas, die in einer zwingend zu beachtenden Reihenfolge der einzelnen Änderungen resultiert.
Δ-MontiArc [Hab+11a] ist eine Erweiterung der Architekturbeschreibungssprache MontiArc [AJB12], die eine Möglichkeit der Variabilitätsbeschreibung auf Architekturebene durch Delta-Modellierung bietet. In diesem

[6] Es gibt Ansätze, die zum Ziel haben, ausschließlich mit Delta-Modulen auszukommen [SBD11].

Ansatz wird ein Featuremodell zur Beschreibung der Variabilität im *problem space* genutzt. In einem Modell-Delta sind neben der Angabe, welche Architekturelemente wie geändert werden müssen, weitere Informationen gespeichert. Dazu zählt, welche Bedingung die ausgewählte Feature-Konfiguration beim *application engineering* erfüllen muss, damit die Operationen des Deltas angewendet werden. Die betroffenen Features können dabei in einem booleschen Ausdruck genutzt werden, der bezogen auf das Vorkommen in der Feature-Kondiguration zu `true` ausgewertet werden muss, damit die Bedingung erfüllt ist. Das in Abbildung 3.18 gezeigte Delta ist zum Beispiel nur relevant, wenn das Feature *Rain_Sensor* konfiguriert wurde.




```
1  delta DRainSensor when Rain_Sensor {
2
3   expand autoconnect;
4
5   modify component IntervalControl {
6      add port in RainSensorStat;
7      add component RainEval;
8    };
9
10   modify component IntervalCmdProcessor {
11     add port in RainIntensity;
12   };
13
14 }
```

Abbildung 3.18: Beispiel eines Feature-Diagramms (oben) und einer entsprechenden Delta-Beschreibung (Bildquelle: [Hab+11a])

Da Δ-MontiArc die Kaskadierung von Deltas ermöglicht, kann als weitere Bedingung angegeben werden, welche anderen Deltas bereits angewendet sein müssen beziehungsweise nicht zuvor angewendet sein dürfen. Über diese Bedingungen lässt sich bei der Produkt-Instanziierung eine korrekte Anwendungsreihenfolge für die ausgewählten Deltas berechnen (vorausgesetzt, es existiert eine Lösung). Alle gültigen Anwendungsreihenfolgen von Deltas einer Produktlinie lassen sich durch eine Baumstruktur darstellen. In Abbildung 3.19 wird eine solche Struktur exemplarisch für eine Soft-

wareproduktlinie dargestellt, deren *solution space*-Variabilität durch eine Basisarchitektur und vier Deltas beschrieben ist.

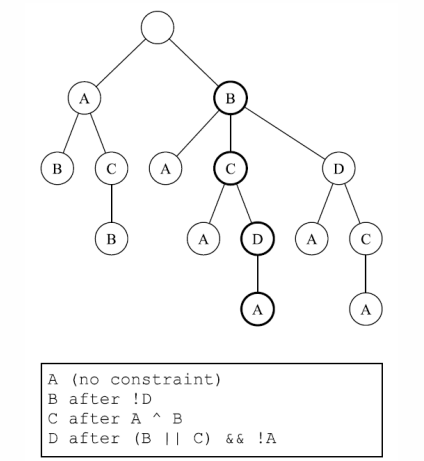


Abbildung 3.19: Mögliche Anwendungsreihenfolgen von vier Deltas einer Produktlinie (Bildquelle: [Hab+11b])

Delta-Modellierung ermöglicht durch die gezielte Modellierung der Unterschiede zweier Systeme eine modulare Beschreibung der Variabilität im *solution space*. Durch die Kaskadierung der Delta-Anwendung ist es sogar möglich, „relativ“ kleine Module zu erzeugen, denn größere Unterschiede lassen sich durch mehrere kleine Deltas (die sich auch überlappen können) beschreiben. Da grundsätzlich alle Architekturelemente modifiziert werden können, bietet dieser Ansatz bezogen auf die verschiedenen Formen der Strukturvariabilität die gleiche Flexibilität wie auch annotative Ansätze. Die Kaskadierungsmöglichkeit erlaubt eine mehrfache Änderung eines Elements. Des Weiteren kann ein Feature in Anwendungsbedingungen mehrerer Deltas verknüpft werden. Die Abbildung zwischen Elementen des *problem space* und denen des *solution space* entspricht also einem *n-to-m mapping*.

Allerdings werden diese Vorteile zugunsten eines Nachteils erkauft: die starke Abhängigkeit zwischen den Modulen. Gerade bei Softwareproduktlinien großer Systeme mit einer starken Variantenvielfalt in allen Formen, wird eine effiziente Modellierung ohne Kaskadierung schwierig, da die zu erwartende durchschnittliche Größe der Module zunehmen würde. Ein Nachteil davon wird beispielsweise bei der Weiterentwicklung einer Produktlinie

sichtbar. Dabei ist es denkbar, dass ein Delta modifiziert werden muss, das sich im Delta-Abhängigkeitsgraphen nah an der Wurzel befindet. Das führt dazu, dass die Deltas in allen Unterbäumen mit dem geänderten Delta als Wurzel gegebenenfalls angepasst werden müssen.

3.5 Zusammenfassung

In diesem Kapitel wurde zuerst ein Beispiel einer Softwareproduktlinie vorgestellt, die ein hohes Maß an Variabilität aufweist. Obwohl das Beispiel keine echte Automobil-Steuergeräte-Software repräsentiert, ist beim Design darauf geachtet worden, dass insbesondere die Variabilitätsaspekte realistisch sind.
Die Variabilität tritt grundsätzlich in drei verschiedenen Formen auf. Konfigurationsvariabliät ist gegeben, wenn eine Komponente über mehrere interne Konfigurationen verfügt, die jeweils eine spezielle Erscheinungsform der Komponente zur Laufzeit repräsentieren. Komponentenvariabilität entsteht durch die Verwendung unterschiedlicher Komponenten an derselben Stelle im System. Die dritte Form ist die Systemstrukturvariabilität. Hierbei unterscheiden sich Systeme hinsichtlich der Beziehungsgeflechte, die durch Links zwischen den Komponenten aufgespannt werden.
Das hohe Maß an Komplexität, das durch Auftreten dieser Variabilitätsformen (und beliebiger Kombinationen) in einer Softwareproduktlinie entsteht, erschwert die Entwicklung einzelner Produkt-Architekturen. Die Forschungsfrage adressiert dieses Problem und definiert das Kernziel dieser Dissertation. Es muss ein Konzept entworfen werden, mit dem Architekten bei der Ableitung von Architekturen in einer Softwareproduktlinie unterstützt werden können.
Der letzte Teil dieses Kapitels befasste sich mit verwandten Arbeiten, um zu zeigen, wie aktuelle Konzepte die Problematik adressieren. Dabei zeigte sich, dass sich mit einigen Konzepten die verschiedenen Variabilitätsformen darstellen und Architekturen ableiten lassen. Allerdings kann zusammenfassend auch festgestellt werden, dass diese Möglichkeit zu Lasten der Skalierbarkeit, Flexibilität oder Änderbarkeit geht.

Kapitel 4

Der fragmentbasierte Lösungsansatz

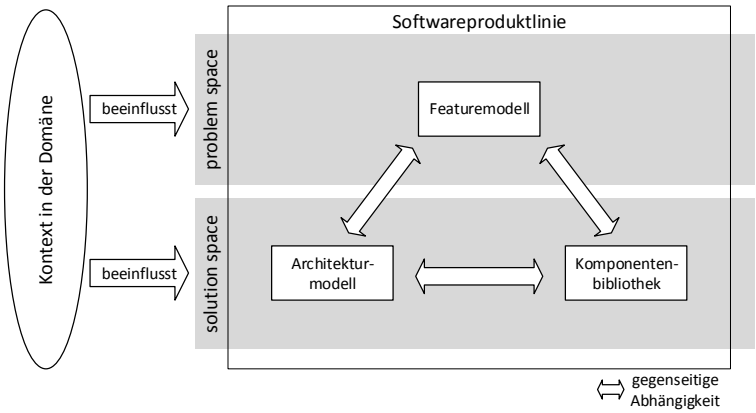


Abbildung 4.1: Die Bausteine und Zusammenhänge einer Softwareproduktlinie

Das erste Ziel, das in dieser Dissertation erreicht werden soll, ist die Definition eines Entwicklungsprozesses (s. Seite 43). Dieser basiert in seiner Grobstruktur auf dem Prozess, den Kang in seinem Konzept *Feature Oriented Reuse Method (FORM)* [Kan+98] eingeführt hat. Kang etablierte mit seiner Arbeit einige grundlegende Konzepte und Begriffe. Diese Konzepte sind auch in den Ansatz dieser Dissertation übernommen. Dazu gehört zum Beispiel die Unterteilung des Entwicklungsprozesses in das *domain engineering* und *application engineering*. Auch die strikte Trennung von

Variabilitätsmodellen in ein Featuremodell und Softwareartefakte basiert auf seiner Arbeit[1].
Eine Besonderheit des fragmentbasierten Ansatzes ist die Beschreibung der Systemstrukturvarianten und der Konfigurationsvarianten, die in einer Softwareproduktlinie existieren können, in unterschiedlichen Modellen. Durch eine geeignete Verknüpfung dieser Modelle werden alle identifizierten Formen der Variabilität im *solution space* unterstützt (s. Abbildung 4.1). Das Architekturmodell beschreibt mit Strukturfragmenten die möglichen Beziehungsgeflechte in der Produktlinie. Die Variabilität in der Systemstruktur kann mit dem Architekturmodell beschrieben werden. In der Komponentenbibliothek sind alle Komponenten der Softwareproduktlinie inklusive der Beschreibung der nutzbaren Komponenten-Konfigurationen enthalten. Mit diesem Wissen wird die Variabilität auf Ebene der Komponenten-Konfigurationen realisierbar. Die Variabilität durch Komponentenwahl lässt sich durch eine geeignete Modellierung der Abhängigkeiten zwischen den Teilen im *solution space* realisieren. Diese Modellierung basiert auf dem in [KHR14] beschriebenen Ansatz.
Die in Abschnitt 3.2.2 genannten Ursachen der *solution space*-Variabilität werden auf zweierlei Weise berücksichtigt. Die Abhängigkeit vom *problem space* wird durch explizite Modellierung einer Beziehung zum jeweiligen Feature des Featuremodells realisiert. Die Abhängigkeit vom Kontext in der Domäne erfolgt indirekt durch alternative Realisierungen von Features.
Die Verknüpfung der einzelnen Modelle erfüllt noch einen weiteren Zweck. Durch sie wird ein einziges Modell der Softwareproduktlinie erstellt, aus dem sich alle Produkt-Architekturen ableiten lassen. Das Modell der Softwareproduktlinie entspricht also einem 150%-Modell (vgl. Abschnitt 3.4.1) mit überlagerten Varianten. Für die Ableitung einer Produktarchitektur wird sowohl das Prinzip der *negativen Variabiltät* wie auch das der *positiven Variabilität* angewandt (s. Abschnitt 2.2). Die für die Realisierung benötigten Module werden aus dem Softwareproduktlinienmodell herausgefiltert und zu Produkt-Architekturen zusammengesetzt.
Im weiteren Teil dieses Kapitels wird zuerst der Entwicklungsprozess vorgestellt. Im Anschluss daran wird gezeigt, wie die Modelle aufgebaut sind, die in diesem Prozess erstellt werden müssen. Die Beschreibung der Beziehungsmodellierung wird im dritten Teil des Kapitels gezeigt. Den Abschluss bildet die Information, wie im *application engineering* eine konkrete Architektur abgeleitet werden kann.

[1] In FORM wurden Features auch zur Beschreibung technischer Aspekte genutzt. In dieser Arbeit beschreiben Features ausschließlich fachliche Aspekte. Statt der in FORM genutzten Bezeichnungen *feature space* und *artifact space* werden in dieser Arbeit die Begriffe *problem space* bzw. *solution space* genutzt.

4.1 Der Entwicklungsprozess

Der Entwicklungsprozess, der in Abbildung 4.2 schematisch dargestellt ist, besteht wie der *FORM*-Prozess aus zwei Hauptschritten: dem *domain engineering* und dem *application engineering*. Im Schritt des *domain engineering* werden die Variabilität der Produktlinie erfasst und die wesentlichen Artefakte für die Erstellung einer Software entwickelt.

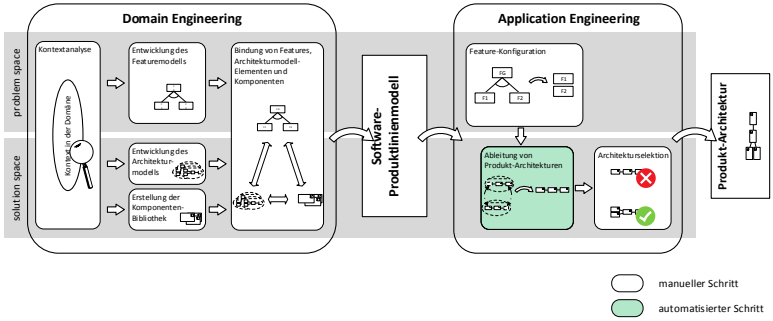


Abbildung 4.2: Schematische Darstellung des Entwicklungsprozesses

Dabei werden schon die fachlichen Aspekte des *problem space* und die softwaretechnischen Aspekte des *solution space* getrennt berücksichtigt. Das Ergebnis des *domain engineering*s ist ein Modell der gesamten Softwareproduktlinie, das beide Bereiche *problem space* und *solution space* umfasst.
Aus diesem Modell werden im *application engineering* einzelne Architekturen abgeleitet. Bei diesem Entwicklungsschritt werden sowohl die fachlichen wie auch die technischen Anforderungen an das Produkt berücksichtigt.

4.1.1 Domain Engineering

Der erste Schritt, der im *domain engineering* durchzuführen ist, ist die **Kontextanalyse**. In dieser Phase wird der Fokus in der Domäne festgelegt, der in der Softwareproduktlinie berücksichtigt werden soll [Kan+98]. Dabei wird bestimmt, was die Produkte der Softwareproduktlinie grundsätzlich leisten sollen und in welchen Umgebungen sie eingesetzt werden. Die Informationen, die in diesem Schritt gesammelt werden, beeinflussen maßgeblich sowohl den *problem space* als auch den *solution space*.

Die Trennung zwischen *problem space* und *solution space* kann genutzt werden, um die verschiedenen Sichten auf die Softwareproduktlinie unabhängig zu modellieren. Domänen-Experten (unter anderem die Kunden), die zu der Gestaltung der Produktlinie bzw. einzelner Produkte beitragen, sind primär an den fachlichen Merkmalen des *problem space* interessiert. Nach der Kontextanalyse erfolgt die **Entwicklung eines Featuremodells**. In diesem wird festgehalten, welche wesentlichen fachlichen Merkmale (Features) in der Softwareproduktlinie auftreten können und welche Kombinationen dieser Merkmale die einzelnen Produkte haben dürfen.

Dagegen ist die Aufgabe der Software-Entwickler gerade die Umsetzung der Produktlinie in Software. Sie stellen die einzelnen Artefakte bereit, aus denen die Software-Produkte erstellt werden können.

Die **Entwicklung des Architekturmodells** findet unabhängig von der **Erstellung der Komponenten-Bibliothek** statt. Dafür gibt es zwei entscheidende Gründe. Zum einen muss berücksichtigt werden, dass die Entwicklung durch unterschiedliche Teams durchgeführt wird. Zum anderen sind der jeweils fokussierte Ausschnitt und der Abstraktionsgrad der Beschreibung unterschiedlich, weshalb entsprechend unterschiedliche Sichten sinnvoll sind. Beispielsweise wird das Architekturmodell des in Abschnitt 3.1 beschriebenen Beispiels von Architekten des Fahrzeugherstellers entwickelt. Das Ergebnis sind Beschreibungen von Beziehungsgeflechten, die für die Komposition der Komponenten herangezogen werden. Die konkreten Funktionalitäten dieser Komponenten werden dagegen häufig von spezialisierten Expertenteams im Betrieb oder von Zulieferern entwickelt. Für die Architektur spielt die konkrete Implementierung eine untergeordnete Rolle. Nur die „von außen“ sichtbaren Eigenschaften – die Konfigurationen der Komponenten – sind interessant.

Um eine Architektur eines konkreten Produkts automatisch durch die Auswahl von Features ableiten zu können, müssen die Abhängigkeiten zwischen den einzelnen Modellen (s. Abbildung 3.1) bekannt sein. Durch die Modellierung der **Bindung von Features, Architekturmodell-Elementen und Komponenten** werden die Abhängigkeiten explizit definiert. Das Ergebnis aus diesem Schritt ist das Softwareproduktlinienmodell.

Kontextanalyse

In Abschnitt 3.1.1 wird gezeigt, dass der Kontext in der Domäne sowohl den *problem space* als auch den *solution space* beeinflusst. Die Kontextanalyse hat zum Ziel die Einflüsse zu erfassen, die bei der Erstellung der Modelle relevant sind.

Dazu werden die Grenzen des Domänenausschnitts definiert, der durch die Softwareproduktlinie abgedeckt werden soll. Die relevanten Aspekte in der Umgebung werden analysiert. Mit dem Verständnis für die physikalischen und technischen Vorgänge und Eigenschaften kann die grundsätzliche Aufgabe spezifiziert werden, die die Produktlinie erfüllen sollen. Diese Aufgabe ist eher abstrakt formuliert. Im vorgestellten Beispiel umschließt die zu lösende Aufgabe die Längsdynamikregelung des Fahrzeugs und die Abgasaufbereitung. Jedes Produkt der Produktlinie hat diese Aufgabe gleichermaßen zu erfüllen. Dafür werden alle Möglichkeiten der Interaktion mit der Umgebung erfasst.
Dabei lassen sich technische Randbedingungen identifizieren, die zum Beispiel zu *feature interaction*-Problemen führen können. Auch rein technische Randbedingungen, die sich nicht auf unterschiedliche Features abbilden lassen, werden erfasst. Die Montage-Reihenfolge der physikalischen Komponenten im Abgasstrang (s. Abschnitt 3.2.2) ist hierfür ein Beispiel.

Entwicklung des Featuremodells

Das Ziel der Featuremodell-Entwicklung ist die Schaffung eines Variabilitätsmodells für den *problem space*. Dafür wird die in der Kontextanalysephase begonnene Analyse der Domäne auf rein fachlicher Ebene fortgesetzt. Innerhalb des fokussierten Teils der Domäne werden einzelne Features identifiziert. Die Features erhält man durch Abstraktion der für die Domäne wesentlichen fachlichen Merkmale der einzelnen Produkte. Diese Abstraktion wird durch die Domänen-Experten durchgeführt und dient unter anderem auch dazu, ein gemeinsames Verständnis (und Vokabular) für alle an der Entwicklung beteiligten Personen zu schaffen.
Auf der Menge der Features wird eine Variabilitätsanalyse durchgeführt. Gemeinsamkeiten und Unterschiede zwischen den verschiedenen Produkten der Produktlinie werden dabei identifiziert. Die Variabilität im *problem space* der Domäne wird in einem Featuremodell festgehalten. Das Featuremodell beschreibt alle in der Softwareproduktlinie erlaubten Kombinationen von Features. Diese Kombinationen werden Feature-Konfigurationen genannt.

Entwicklung des Architekturmodells

Auf die Analyse des Kontextes folgt auf der technischen *solution space*-Ebene die Entwicklung der Software-Artefakte. Das Architekturmodell beschreibt die Beziehungsgeflechte zwischen den Komponenten der Produktlinie. Es dient gleichzeitig als Variabilitätsmodell für die Verbindungs-

strukturen im *solution space*. Dazu müssen die Gemeinsamkeiten und Unterschiede hinsichtlich dieser Strukturen zwischen den verschiedenen Produkten identifiziert und im Architekturmodell beschrieben werden.

Die wichtigsten Prinzipien, die bei der Entwicklung des Architekturmodells berücksichtigt werden, sind Abstraktion und Modularisierung. Abstraktion wird durch Weglassen konkreter Softwarekomponenten aus der Strukturbeschreibung erreicht. Diese Struktur repräsentiert sozusagen eine „Lochmaske“, in dessen Löcher bestimmte Komponenten passen. Für die Modularisierung werden die einzelnen Systemstrukturen zerlegt in Strukturfragmente, die jeweils für die Komposition verschiedener Systeme wiederverwendet werden können.

Die Abstraktion von konkreten Komponenten eignet sich gut für die Entwicklung eingebetteter Steuerungssystemen wie sie in der Automobil-Industrie eingesetzt werden. Diese Systeme realisieren häufig ähnliche Steuerungs- und Regelungsaufgaben, weshalb sich die Struktur der Systeme ähnelt. Verschiedene strukturelle Architektur- und Designmuster werden an unterschiedlichsten Stellen eingesetzt. Solche Muster lassen sich zum Beispiel gut auf abstrakte Strukturfragmente abbilden.

Für die Definition einzelner Strukturfragmente müssen geeignete Grenzen identifiziert werden. Für diese Identifizierung werden Features herangezogen. Es gilt: jedes Strukturfragment beschreibt den Ausschnitt der Softwarearchitektur, der mindestens ein Feature realisieren kann.

Erstellung der Komponenten-Bibliothek

Die Erstellung der Komponenten-Bibliothek untergliedert sich in zwei Phasen. Zum einen ist das die eigentliche Entwicklung der Komponente. Zum anderen werden die Komponenten hinsichtlich ihrer Variabilität analysiert und mit den entsprechenden Ergebnissen in einer Bibliothek zusammengetragen.

Wie die jeweilige Komponente konkret entwickelt wird, spielt wie gesagt bei diesem Ansatz keine Rolle. In dieser ersten Phase müssen allerdings die Ports spezifiziert werden, über die eine Interaktion mit der Komponente ermöglicht wird.

In der für den Ansatz interessanteren zweiten Phase des Prozessschritts werden die Konfigurationen der Komponenten festgelegt. Jede Konfiguration repräsentiert genau eine Funktionalität und kapselt die Menge der Ports, die für die Realisierung der jeweiligen Funktion benötigt wird. Mehrere Konfigurationen in einer Komponente definieren die Variabilität dieser Komponente.

Die Komponenten-Bibliothek fasst die Beschreibungen *aller* Komponenten der Produktlinie zusammen.

Bindung von Features, Architekturmodellelementen und Komponenten

Im letzten Schritt des *domain engineering*s werden die verschiedenen unabhängig voneinander entwickelten Modelle miteinander zu einem einzigen Produktlinienmodell verbunden.
Der Architekt bestimmt für jedes Feature aus dem Featuremodell mögliche Implementierungen. Dafür wählt er ein Strukturfragment und eine passende Menge von Komponenten aus. Durch die Bindung der Komponenten an die Bestandteile des Strukturfragments wird eine Komposition beschrieben, die das entsprechende Feature implementiert. Um technische Randbedingungen, die bei der Kontextanalyse festgestellt wurden, zu berücksichtigen, kann der Architekt mehrere unterschiedliche Implementierungen für ein Feature erstellen.
Alle Features müssen auf diese Weise mit Implementierungen in Beziehung gesetzt werden. Sind dabei alle technischen Randbedingungen berücksichtigt, erhält man eine Menge von Softwaremodulen. Aus diesen Modulen lassen sich alle Produkte der Softwareproduktlinie zusammensetzen.

4.1.2 Application Engineering

Beim *application engineering* werden aus dem Software-Produktlinienmodell konkrete Softwarearchitekturen abgeleitet. Die Variabilität, die in den Variabilitätsmodellen aus dem *domain engineering* enthalten ist, wird unter Berücksichtigung einer ausgewählten Feature-Konfiguration aufgelöst. Das bedeutet, dass aus den verschiedenen Varianten, die existieren, genau eine ausgewählt wird.
Bei der Entscheidung, welche Variante gewählt wird, müssen sowohl die fachlichen als auch die technischen Randbedingungen berücksichtigt werden. Die Ableitung der Architektur aus dem Software-Produktlinienmodell erfolgt automatisiert.

Feature-Konfiguration

Bei dem Prozessschritt Feature-Konfiguration legt ein Domänen-Experte (zum Beispiel der Kunde) die Menge Features fest, die das gewünschte Software-Produkt erfüllen soll. Da bei der Auswahl der Features ausschließ-

lich eine Entscheidung bezüglich fachlicher Anforderungen der Software getroffen wird, ist dieser Schritt dem *problem space* zugeordnet.
Im Featuremodell wird durch Angabe entsprechender Randbedingungen festgelegt, welche Kombinationen von Features in der Produktlinie möglich sind. Dadurch ist das Ergebnis dieses Schritts immer eine Menge von Features, die ein valides System der Produktlinie beschreibt. Diese Feature-Menge wird auch Feature-Konfiguration genannt. Der Begriff Feature-Konfiguration bezeichnet homonym den Prozessschritt und die ausgewählte Feature-Menge.

Ableitung von Produkt-Architekturen

Die Ableitung von Produktarchitekturen erfolgt teilautomatisiert aus dem Software-Produktlinienmodell nach der Vorgabe einer Feature-Konfiguration. Das entsprechende Produkt muss selbstverständlich genau die angegebenen Features besitzen.
Die modellierten Beziehungen zwischen *solution space* und *problem space* im Software-Produktlinienmodell werden genutzt, um alle Softwareteile herauszufiltern, die sich für die Realisierung eines Produkts eignen. Die Bindungen zwischen den Strukturfragmenten und den Komponenten erlauben die Erstellung von Architekturmodulen. Diese Module werden zu Softwarearchitekturen zusammengesetzt.
Durch die Vorgabe einer Feature-Konfiguration können die fachlichen Einflüsse auf die Auflösung der *solution space*-Variabilität berücksichtigt werden. Der Kontext der Domäne beeinflusst aber auch direkt den *solution space*. Die Verknüpfung von Software-Artefakten zu diesen technischen Randbedingungen ist im Software-Produktlinienmodell nicht enthalten. Daher ist es erforderlich, dass alle alternativen Produktarchitekturen erzeugt werden, um sicher zu gehen, dass die tatsächlich benötigte gefunden werden kann. Alle erstellten Produktarchitekturen werden dem Entwickler vorgeschlagen.

Architekturselektion

Im letzten Schritt des Prozesses wählt der Entwickler aus dieser Menge unterschiedlicher Architekturen die passende aus.
Es ist möglich, dass in der Menge dieser Architekturen auch Strukturen enthalten sind, die kein valides System repräsentieren. Diese muss der *application-engineer* identifizieren und verwerfen. Damit diese Lösung nicht bei jeder Produktableitung erneut vorgeschlagen wird, muss er sie als invalide kennzeichnen.

Bei der Entscheidung, welche der grundsätzlich validen Architekturen zum Einsatz kommt, fließen dann die technischen Randbedingungen ein. In der Beispielproduktlinie sind das zum Beispiel die Montagereihenfolge der Hardware-Komponenten im Abgassystem und besondere Lösungen zur Vermeidung von Feature-Interaction-Problemen.

Am Ende des Prozesses entscheidet sich der Entwickler für genau eine Lösung. Diese Lösung repräsentiert die Produkt-Architektur, die alle Anforderungen (fachliche wie technische) erfüllt.

4.2 Modellierung der Artefakte

Der vorgestellte Prozess beschreibt, dass verschiedene Aspekte der Softwareproduktlinie unabhängig voneinander in eigenen Schritten modelliert werden. Das Konzept sieht vor, dass die einzelnen Teile (Featuremodell, Architekturmodell und Komponenten-Bibliothek) in eigenen Modellen entwickelt werden. Zusätzlich wird beschrieben, wie diese drei Einzelteile zu einem Modell der Softwareproduktlinie zusammengesetzt werden können. Das erfordert, dass das Metamodell, das die Modellbeschreibung für die Softwareproduktlinie definiert, eine solche unabhängige Entwicklung ermöglicht.

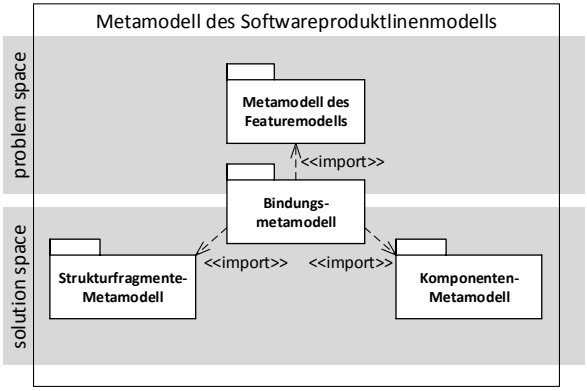


Abbildung 4.3: Bestandteile des Softwareproduktlinien-Metamodells

Das Metamodell besteht grob aus vier Teilen (s. Abbildung 4.3). Für das Featuremodell, Architekturmodell und die Komponentenbibliothek wird jeweils ein eigenes Teil-Metamodell definiert. Die Abhängigkeiten zwischen den Modellen beschreibt ein zusätzlicher Teil. Dieser definiert, wie die Bindungen zwischen den einzelnen Artefakten modelliert werden können.
Im folgenden Abschnitt wird zunächst geklärt, was die einzelnen Modellbestandteile leisten müssen. Dabei wird der Blick insbesondere auf die Modellierung der Variabilität gerichtet. Für die Modellierung der Systemstruktur-Variabilität werden Strukturfragmente genutzt. Die Konfigurations-Variabilität wird durch das Komponentenmodell ermöglicht. Durch geeignete Bindungen wird die Komponenten-Variabilität realisiert.
Im letzten Teil dieses Abschnitt wird beschrieben, wie das Softwareproduktlinienmodell im *application engineering* genutzt werden kann. Es wird ein Verfahren vorgestellt, mit dem Softwarearchitekturen für einzelne Produkte abgeleitet werden können.

4.2.1 Featuremodell

In Abschnitt 3.1.2 wurde bereits ein Beispiel für ein Featuremodell gezeigt, das alle für das Konzept notwendigen Eigenschaften hat. Darum wird diese Modellierungsweise für die Modellierung der Varibilität im *problem space* in dieser Dissertation übernommen. Im folgenden Abschnitt werden die wesentlichen Eigenschaften näher beschrieben.
Dafür wird zuerst erklärt, wie ein Featuremodell aufgebaut ist. Da dieses Modell als Variabilitätsmodell des *problem space* fungiert, wird im darauf folgenden Abschnitt erklärt, welche Möglichkeiten der Variabilitätsbeschreibung das Featuremodell haben muss. Abschließend muss noch beschrieben werden, wie eine Feature-Konfiguration aussieht, die im *application engineering*-Schritt erstellt wird.

Features

Die wesentlichen Elemente eines Featuremodells sind natürlich die Features. Features repräsentieren Merkmale eines Softwaresystems auf abstrakte Weise. Einzelne Features können in Hierarchiebeziehungen angeordnet sein. Das erhöht unter anderem die Übersichtlichkeit in der Darstellung, die üblicherweise in einer Baumstruktur erfolgt.
Durch diese Hierarchien können Features gruppiert und klassifiziert werden. Hierarchisierung bewirkt auch, dass Features Merkmale auf unterschiedlichen Abstraktionsebenen repräsentieren. *Feature-Klasse 2* in Abbildung 4.4 ist abstrakter als die ihm zugeordneten Features *ODER-Feature 1*

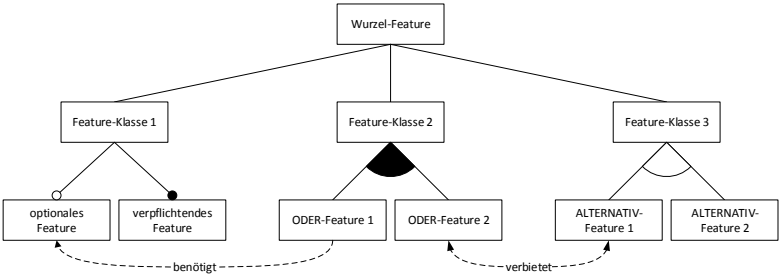


Abbildung 4.4: Die unterstützten Modellierungs-Möglichkeiten der *problem space*-Variabilität in einem Diagramm dargestellt

und *ODER-Feature 2*. Gleichzeitig stehen diese „Kindfeatures“ implizit in einer „Geschwister“-Beziehung zueinander, die genutzt wird, um eine Feature-Gruppe zu bilden. Diese Klassifizierung ermöglicht, Features zum Beispiel aus unterschiedlichen *Subdomänen* voneinander abzugrenzen.

Das Featuremodell als Variabilitätsmodell

Die Variabilität, die das Featuremodell auf *problem space*-Ebene darstellen soll, kann auf drei unterschiedliche Weisen erfolgen. Zum einen verfügt jedes Feature über eine Optionalitäts-Eigenschaft. Das bedeutet, dass ein Feature entweder optional oder verpflichtend ist. Optional bedeutet, dass dieses Feature nicht in allen Feature-Konfigurationen enthalten ist. Entsprechend verfügen die jeweiligen Systeme der Produktlinie nicht über dieses Merkmal. Im Gegensatz dazu sind Features, die in allen Feature-Konfigurationen vorhanden sind, im Featuremodell als verpflichtend (engl. *mandatory*) deklariert.
Für eine zweite Möglichkeit der Variabilitätsmodellierung können Feature-Gruppen genutzt werden. Die Gemeinsamkeit der Features einer Feature-Gruppe ist eine Beziehung zum selben Elternknoten in der Baumstruktur des Featuremodells. Die Zuordnung eines logischen Operators zur Gruppe ermöglicht eine Einschränkung der Menge an unterschiedlichen Kombinationen von Features, die in Feature-Konfigurationen jeweils auftreten können. In dieser Dissertation werden nur die logischen Operatoren *oder* (OR) und *exklusiv oder* (XOR) verwendet. Das bedeutet für eine *OR-Feature-Gruppe*, dass mindestens eins der entsprechenden Features in der

Feature-Konfiguration enthalten sein muss. Bei einer *XOR-Feature-Gruppe* ist immer genau ein Feature ausgewählt. Alle Features einer Feature-Gruppe müssen zwangsläufig optional sein, weshalb eine explizite Anzeige dieser Feature-Eigenschaft nicht gemacht wird.

Die dritte Möglichkeit zur Variabilitätsmodellierung wird durch die Spezifikation von Querbeziehungen realisiert. Die Beziehungen, die damit zwischen zwei beliebigen Features des Featuremodells hergestellt werden, schränken die Anzahl möglicher Feature-Konfigurationen weiter ein. Die Existenz eines Features in einer Feature-Konfiguration kann über eine Querbeziehung von einem anderen Feature abhängig gemacht werden. In dieser Dissertation werden zwei Arten von Querbeziehungen genutzt: *requires* und *mutex*.

Über eine gerichtete *requires*-Beziehung kann ausgedrückt werden, dass das Zielfeature benötigt wird, wenn das entsprechend andere in der Feature-Konfiguration enthalten sein soll. Beim Featuremodell im Bild 4.4 bedeutet das, dass das Feature *optionales Feature* immer in der Feature-Konfiguration enthalten ist, wenn auch das *ODER-Feature 1* ausgewählt ist. Die Optionalitäts-Angabe wird dadurch deutlich eingeschränkt, denn das Feature *optionales Feature* ist nicht mehr optional, wenn in der Feature-Konfiguration das *ODER-Feature 1* bereits gewählt ist. Die andere unterstützte Querbeziehung schließt ein gleichzeitiges Vorkommen der entsprechenden Features in allen Feature-Konfigurationen aus. Diese *mutex*-Beziehung (kurz für *mutual exclusive*) wirkt bidirektional.

Durch diese explizit modellierten Querbeziehungen ist der Graph, mit dem das Featuremodell repräsentiert wird, kein Baum. Dennoch wird es weiterhin mit dieser Grundstruktur dargestellt, um die oben genannten Vorteile – die Klassifizierung – sichtbar zu machen.

Feature-Konfigurationen

Im *application engineering* Prozess wird durch den Domänenexperten durch Angabe einer Feature-Konfiguration festgelegt, welche Eigenschaften die Software haben soll. Jede Feature-Konfiguration genügt den Regeln des Featuremodells. Das heißt, es sind alle verpflichtenden Features enthalten, und die Einschränkungen hinsichtlich der Feature-Gruppen-Eigenschaft und der Querbeziehungen sind berücksichtigt. In einer Feature-Konfiguration existiert keine Variabilität auf fachlicher Ebene.

Um die Vorteile der baumartigen Repräsentation des Featuremodells auch bei der Darstellung einer Feature-Konfiguration zu haben, wird die Hierarchisierung aus dem Featuremodell wiederverwendet. Eine Randbedingung

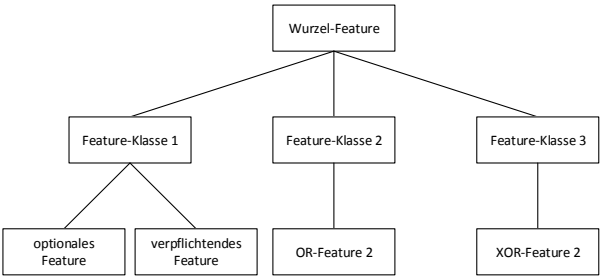


Abbildung 4.5: Darstellung einer bezüglich des in Abbildung 4.4 dargestellten Modells validen Feature-Konfiguration

hierfür ist, dass alle Features entlang des Pfades vom selektierten Feature bis zur Wurzel in der Feature-Konfiguration enthalten sind.
Abbildung 4.5 zeigt eine so dargestellte valide Feature-Konfiguration, die aus dem vorgestellten Featuremodell abgeleitet wurde (vgl. Abbildung 4.4). Durch die Darstellung einer Feature-Konfiguration in Form eines Baums ist die im Featuremodell beschriebene Klassifizierung weiterhin erkennbar.

4.2.2 Architekturmodell

Das Architekturmodell beschreibt die Beziehungsgeflechte, die in den Softwarearchitekturen der Produktlinie zwischen den Komponenten existieren. In der Prozessbeschreibung (s. Abschnitt 4.1) wird erklärt, dass die Entwicklung des Architekturmodells unabhängig von der Komponentenentwicklung durchgeführt wird. Das Architekturmodell muss darum ohne Komponenten beschrieben werden können. Der Abstraktionsgrad, auf dem die möglichen Beziehungsstrukturen beschrieben werden, ist daher sehr hoch.
Das Architekturmodell eignet sich ideal, um die Variabilität auf Systemstrukturebene zu beschreiben. Die Modularisierung der Modellbeschreibung durch die Beschreibung einzelner Strukturfragmente schafft dabei eine Möglichkeit zur Wiederverwendung.
Im folgenden Teil dieses Kapitels wird zunächst geklärt, wie die Elemente des Architekturmodells – die Strukturfragmente – aufgebaut sind und wie diese zueinander in Beziehung stehen. Im Anschluss daran wird

beschrieben, wie mit Hilfe des Architekturmodells die Variabilität auf Systemstrukturebene modelliert wird.

Strukturfragmente

Ein wesentlicher Bestandteil einer Softwarearchitektur sind die Beziehungen der Komponenten zueinander. Durch eine abstraktere Beschreibung des Beziehungsgeflechts kann man gezielt die Unterschiede auf der Systemstrukturebene zeigen. Die Abstraktion wird erreicht, indem auf eine konkrete Angabe der Komponenten verzichtet wird. Ein Strukturfragment beschreibt einen Ausschnitt dieses Beziehungsgeflechts.

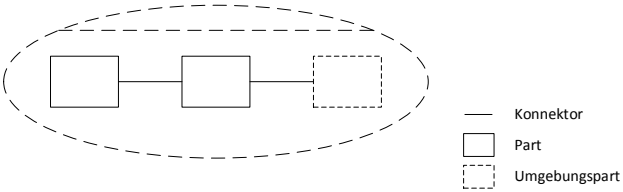


Abbildung 4.6: Ein Strukturfragment besteht aus Parts und Konnektoren

Die Elemente, aus denen ein Strukturfragment aufgebaut ist, sind *Parts* und *Konnektoren*. Mit diesen wird ein Netz aufgespannt, bei dem die Knoten durch Parts und die Kanten durch Konnektoren gebildet werden (s. Abbildung 4.6).
Strukturfragmente sind die Module, aus denen die Beziehungsgeflechte der Produktarchitekturen zusammengesetzt sind. Das Besondere an Strukturfragmenten ist die Art wie sie zu einer Gesamtstruktur zusammengesetzt werden. Strukturfragmente werden teilweise übereinandergelegt und miteinander zu einer größeren Struktur verschmolzen. Auf diese Weise kann die Struktur des Beziehungsgeflechts jeder Produktarchitektur erzeugt werden.
Wie beim Featuremodell ist es auch im Architekturmodell hilfreich, die Anzahl der möglichen Kombinationen einzuschränken zu können. Hierfür wird jedes Strukturfragment, das nicht beliebig mit anderen Strukturfragmenten verschmolzen werden können soll, durch zusätzliche Informationen angereichert. Diese Information beschreibt Bedingungen, welche die Umgebung des Strukturfragments nach einer Verschmelzung erfüllen muss. Für die Beschreibung der Umgebung existieren spezielle Umgebungsparts.

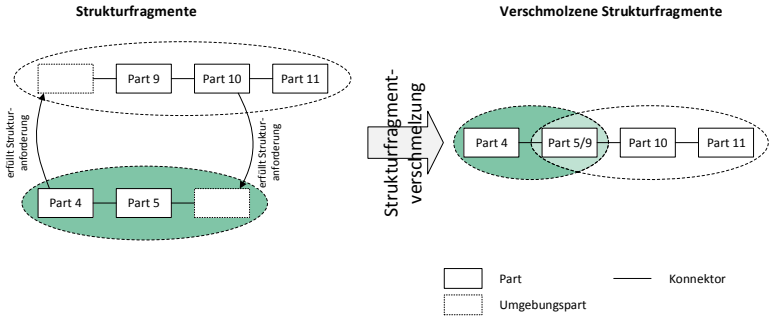


Abbildung 4.7: Verschmelzung zweier Strukturfragemente

Die Form der geforderten Umgebungsstruktur muss durch Parts und Konnektoren des zu verschmelzenden Fragments gebildet werden. Ist die Form identisch, werden die entsprechenden Parts an die Umgebungsparts gebunden. Das bedeutet, dass die Angabe eines Umgebungsparts dazu führt, dass genau an dieser Stelle eine Verschmelzung mit einem anderen Strukturfragment durchgeführt werden muss.
Genauso kann dieses zweite Strukturfragment eine Anforderung an seine Umgebung stellen. Wenn zwei Strukturfragmente gegenseitig die Umgebungsanforderungen erfüllen, können sie miteinander verschmolzen werden. In Abbildung 4.7 wird das Strukturfragment für die Beschreibung der Abgasaufbereitung mit dem für die Antriebsanbindung verschmolzen. Durch *Part 4* und *Part 10* werden die Anforderungen der entsprechenden Umgebungsparts erfüllt. Die Überlappung der Fragmente ist auf der rechten Seite des Bildes zu erkennen.

Strukturfragmente als Graph

Um das Konzept der Strukturfragmente einfacher zu verstehen, kann eine Abstraktion unter Zuhilfenahme der Graphentheorie erreicht werden. Ein Strukturfragment kann durch einen ungerichteten Graphen $G = (V, E, f, g)$ repräsentiert werden, bei dem sowohl Knoten als auch Kanten gefärbt sind. Die Knotenmenge V beschreibt dabei die Parts und Umgebungsparts, die Kantenmenge E die Konnektoren des Strukturfragments.
Wenn die Elemente der Menge C die verschiedenen vorkommenden Farben repräsentieren, lässt sich die Färbung der Knoten durch folgende Abbil-

dung beschreiben: $f : V \rightarrow C \subseteq \mathbb{N}_0$. Die Abbildung $g : E \rightarrow C \subseteq \mathbb{N}_0$ beschreibt analog die Färbung der Kanten. Für die Darstellung von Strukturfragmenten werden in C nur zwei Elemente benötigt. Das Beispiel in Abbildung 4.8 nutzt schwarz zur Färbung von Knoten, die jeweils einen Part des Strukturfragments repräsentieren. Knoten, die für Umgebungsparts stehen, werden grün gefärbt. Kanten werden schwarz gefärbt, wenn die entsprechenden Konnektoren zwei Parts verbinden. Steht ein Ende eines Konnektors mit einem Umgebungspart in Beziehung, wird der entsprechende Konnektor ebenfalls grün gefärbt. An dem Beispiel in der Abbildung lässt sich erkennen, dass die Färbung im Sinne der Graphentheorie keine „gültige Färbung“ ist, denn es existieren adjazente Knoten mit der gleichen Färbung.

Das Architekturmodell einer Softwareproduktlinie lässt sich als Graph G_A durch die Vereinigung aller n Graphen, die je ein Strukturfragment repräsentieren, abstrahiert beschreiben: $G_A = \bigcup_{n \in \mathbb{N}} G_n$.

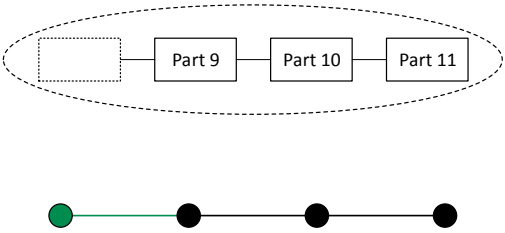


Abbildung 4.8: Abstraktion eines Strukturfragments durch einen knoten- und kantengefärbten Graphen

Die Verschmelzung von Strukturfragmenten lässt sich durch Operationen auf dem Graphen des Architekturmodells G_A veranschaulichen. Abbildung 4.9 zeigt anschaulich die Folge von Operationen, die eine Verschmelzung von zwei Strukturfragmente repräsentierenden Teilgraphen von G_A zur Folge haben.

Zuerst wird eine neue Kante so eingefügt, dass die Knoten, die jeweils inzident zur eingefügten Kante sind, zu Graphen verschiedener Strukturfragmente gehören. Durch Kontraktion dieser neuen Kante (also der Verschmelzung der zu dieser Kante inzidenten Knoten (u, v) mir der anschließenden Entfernung der Kante [Tit03]) entsteht ein zusammenhän-

gender Graph. Zuvor werden so viele zu u bzw. v adjazente, grün gefärbte Knoten entfernt, dass der Grad des durch die Kontraktion entstandenen Knotens w kleiner oder gleich des jeweiligen Grades von u bzw. v ist, bevor die kontrahierte Kante eingefügt wurde. Im Beispiel ist der Grad von u zwei. Der Grad von w ist ebenfalls zwei. Das bedeutet, dass der grün gefärbte zu u adjazente Knoten entfernt werden kann. Das Gleiche gilt für den Knoten v. Wäre der Grad von u zu Beginn drei gewesen, wäre w zu einem grün gefärbten Knoten adjazent.

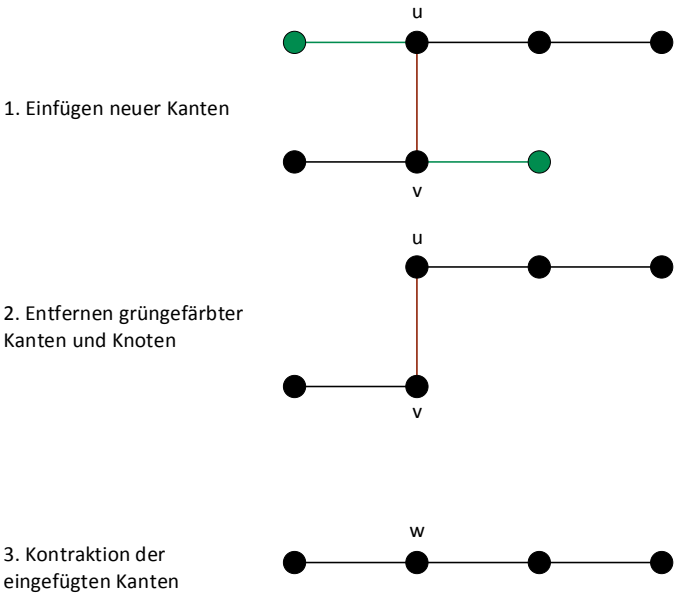


Abbildung 4.9: Verschmelzung zweier Strukturfragmente, beschrieben als Folge von Operationen auf Graphen

Systemstruktur-Variabilität im Architekturmodell

Mit einem Strukturfragment allein lässt sich die Systemstruktur-Variabilität nicht umsetzen. Es beschreibt ja genau eine Variante eines Strukturausschnitts. Erst durch die Verschmelzung unterschiedlicher Fragmente werden verschiedene Beziehungsgeflechte erzeugt.

In der Abbildung 4.10 wird an zwei Architekturausschnitten aus dem Begleitbeispiel (s. Abschnitt 3.1) dargestellt, wie zwei unterschiedliche Systemstrukturen erzeugt werden. Zu sehen sind die Ausschnitte, in denen unterschiedliche Komfortsysteme mit Längsdynamikkoordination und Abgasaufbereitung integriert werden können. Dafür werden jeweils drei Strukturfragmente verschmolzen. Der „Mittelteil“ der Struktur wird bei den beiden Architekturen jeweils durch ein anderes Fragment beschrieben.

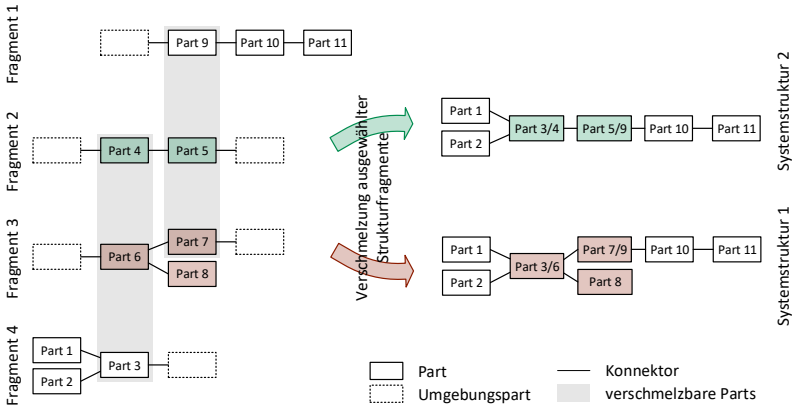


Abbildung 4.10: Ein Beispiel für die Erzeugung von Systemstruktur-Variabilität durch alternative Strukturfragment-Verschmelzungen

Um eine Produktarchitektur automatisch generieren zu können, müssen alle möglichen Beziehungen zwischen den Komponenten bekannt sein. Es darf keine „Lücken“ in der Beschreibung des Beziehungsgeflechts der Produktarchitektur geben. Daraus folgt, dass jede Produktarchitektur hinsichtlich der Struktur ihres Beziehungsgeflechts vollständig durch überlappende Strukturfragmente beschrieben werden muss.

Ein Strukturfragment kann nicht nur für gleiche Strukturen in unterschiedlichen Produktarchitekturen wiederverwendet werden. Aufgrund der Abstraktion von den Komponenten lassen sich auch gleichförmige Strukturen innerhalb einer Produktarchitektur beschreiben. Da das Architekturmodell alle Strukturfragmente der Produktlinie enthält, repräsentiert es das Strukturmodell für alle Produktarchitekturen. Und somit ist es das *solution space*-Variabilitätsmodell bzgl. der Systemstrukturvariabilität.

4.2.3 Komponenten-Bibliothek

In der Komponenten-Bibliothek werden alle Komponenten gesammelt, die in der Softwareproduktlinie eingesetzt werden dürfen. Neben den Komponenten selbst muss die Information bereitgestellt werden, über welche Konfigurationen die Komponenten verfügen.
In diesem Abschnitt wird erklärt, was eine Komponente leisten muss, um in die Bibliothek aufgenommen werden zu können. Insbesondere wird gezeigt, welche Variabilitätsform man mit einer entsprechenden Komponente modellieren kann.

Komponente

Eine Softwarekomponente kapselt ganz im Sinne der Definition von Szyperski [2] einen Teil der Software in Form einer *blackbox*-Abstraktion.
Die Funktionalität einer Komponente muss auf abstrakte Weise bekannt sein. Das bedeutet, dass man weiß, welches Verhalten eine Komponente aufweisen kann. Darüber hinaus muss ebenfalls bekannt sein, über welche Schnittstellen die Komponente mit der Umgebung interagieren muss, damit sie das entsprechende Verhalten zeigt. Der konkrete innere Aufbau der Komponente ist dabei nicht erkennbar. Softwarekomponenten bilden somit in dieser Arbeit atomare Software-Einheiten.
In dieser Dissertation werden die Schnittstellen durch Ports realisiert. Es werden zwei Arten von Ports eingesetzt, um die komplementären Eigenschaften einer Kommunikationsschnittstelle im Modell explizit zu trennen. Diese Eigenschaften werden zum Beispiel durch den Datenfluss bestimmt. Über Output-Ports kann eine Komponente Daten nach außen geben. Input-Ports nutzt sie, um Daten von außen zu bekommen. Für die Interaktion zwischen Komponenten müssen im Allgemeinen weitere, nicht-komplementäre Eigenschaften von Ports bekannt sein.
Hierzu zählt zum Beispiel das Interaktionskonzept, das an dem entsprechenden Port umgesetzt ist – also beispielsweise, ob Nachrichten synchron oder asynchron ausgetauscht werden. Für die Entwicklung des Ansatzes in dieser Arbeit reicht es zu wissen, ob Ports, über die eine Interaktion realisiert wird, kompatibel sind. Mehrere Ports gelten als kompatibel, wenn alle nicht komplementären Eigenschaften identisch sind. Im weiteren Verlauf wird die Menge aller Port-Eigenschaften eineindeutig auf einen Port-Typen abgebildet. Durch diese Abstraktion lässt sich Kompatibilität von Ports durch die Gleichheit ihrer Typen feststellen.

[2] "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. [...]" [Szy98]

Variabilität durch Komponenten-Konfigurationen

Eine Konfiguration einer Komponente bestimmt einerseits das Verhalten der Komponente und andererseits die Ports, die für die Realisierung dieses Verhaltens benötigt werden. Variabilität in einer Komponente kann beide Aspekte einer Konfiguration betreffen. Aus diesem Grund ist es sinnvoll, die Variabilität von Komponenten auf Ebene von Konfigurationen zu beschreiben. Die Konfiguration einer Komponente legt von außen sichtbare Eigenschaften einer Komponente im Sinne der Softwarearchitektur-Definition auf Seite 8 fest. Die verschiedenen für die Softwareproduktlinie relevanten Varianten dieser Eigenschaften werden durch unterschiedliche Komponenten-Konfigurationen repräsentiert.

Verschiedene Verhaltensweisen sind sinnvoll, wenn die Komponente für die Realisierung mehrerer Features zuständig ist, die nicht immer gleichermaßen im Produkt enthalten sein sollen. Für die Realisierung unterschiedlicher Verhaltensweisen wird eine Komponente verschiedenartige Daten verarbeiten und bereitstellen. Die Mengen der Ports, die mit unterschiedlichen Verhaltensweisen assoziiert sind, müssen sich unterscheiden können.

Unterschiedliche Mengen von Ports können auch eine Folge der Forderung sein, dass eine Komponente in unterschiedlichen Umgebungen eingesetzt werden soll. Wenn die Umgebungen die von der Komponente benötigten Daten auf andere Weise zur Verfügung stellt oder von der Komponente zur Verfügung gestellt bekommt, ist die Integration der entsprechenden Komponente am einfachsten, wenn die Komponente an verschiedene Umgebungen angepasst werden kann. Dabei kann auch das Verhalten betroffen sein.

Es ist wichtig zu verstehen, dass in dieser Dissertation nicht jede mögliche Erscheinungsform einer Komponente hinsichtlich Verhalten und Portmenge eineindeutig auf eine Komponenten-Konfiguration abgebildet wird. Nur die für die Softwareproduktlinie relevanten Varianten müssen unterschieden werden. Soll eine Komponente beispielsweise über eine Adaptionsmöglichkeit zur Laufzeit verfügen wie im DAISi-Konzept [Nie+07], darf eine eineindeutige Abbildung nicht einmal durchgeführt werden. Da die Bindung einer Variante der Komponenten-Konfiguration bereits im *application engineering* durchgeführt wird, stünden alle anderen für die Adaption benötigten Verhaltensweisen zur Laufzeit nicht mehr zur Verfügung.

Sind in mehreren Konfigurationen die gleichen Ports zusammengefasst, sind die Konfigurationen zueinander syntaktisch kompatibel.

Besitzt eine Komponente mehrere Konfigurationen, verfügt sie über die Möglichkeit, die geforderte Konfigurations-Variabilität zu realisieren. Durch

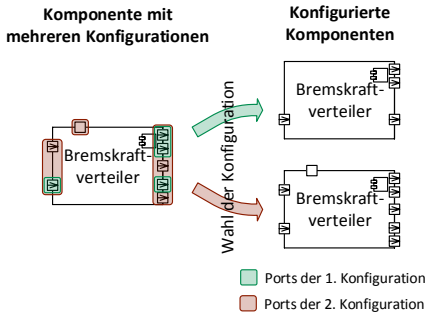


Abbildung 4.11: Konfigurations-Variabilität durch alternative Konfigurationen in Komponenten

die Wahl einer Konfiguration bei der Erstellung der Architektur lässt sich unterschiedliche Funktionalität in verschiedenen Umgebungen integrieren. Abbildung 4.11 zeigt die Nutzung mehrerer Konfigurationen einer Komponente am Beispiel des Bremskraftverstärkers (s. Abschnitt 3.1). Die Bremskraftverstärkerkomponente soll in mehreren Umgebungen mit jeweils unterschiedlichen Funktionalitäten genutzt werden. Durch die Wahl einer Konfiguration wird festgelegt, ob die Komponente zum Beispiel die Bremskraft über zwei oder vier Ventile an die Räder weitergibt. Die Ansteuerung von vier Ventilen ermöglicht die Aktivierung einer ESP-Funktionalität der Bremskraftverteiler-Komponente.
Da diese Auswahl zur Zeit der Architektur-Erstellung getroffen wird, stehen die anderen Ports danach nicht mehr zur Verfügung. Zur Laufzeit sind diese Konfigurationen statisch, das heißt sie können nicht mehr geändert werden. Die Komponente wird deswegen in der Abbildung einer Produktarchitektur vereinfacht dargestellt, indem die nach der Entscheidung nicht mehr nutzbaren Ports ausgeblendet werden.

4.3 Bindung der Artefakte

In den vorherigen Abschnitten wurde gezeigt, wie mit unterschiedlichen Artefakten – dem Featuremodell, dem Architektur-Modell und der Komponenten-Bibliothek – verschiedene Aspekte der Softwareproduktlinie mo-

delliert werden können. Das Softwareproduktlinienmodell muss zusätzlich zu den Artefakten die Abhängigkeiten zwischen diesen beschreiben.
Die Idee bei der Modellierung dieser Abhängigkeiten besteht darin, zuerst eine Bindung zwischen den Artefakten im *solution space* herzustellen. Komponenten werden über *Implementierungsbindungen* an Strukturfragmente gebunden. Damit lässt sich die bislang noch nicht gezeigte Form der *solution space*-Variabilität, die Komponenten-Wahl, realisieren. Die Abhängigkeiten der Software-Artefakte untereinander sind damit aufgelöst, und gleichzeitig sind alle Formen der *solution space*-Variabilität modellierbar.
Über *Feature-Implementierungen* wird festgelegt, welche Implementierungsbindungen für die Realisierung eines Features genutzt werden können. Die Abhängigkeiten zwischen *problem space* und *solution space* werden durch eine Beziehung zwischen dem Featuremodell und den Feature-Implementierungen aufgelöst. Die in Abbildung 4.1 dargestellte Abhängigkeit von Featuremodell zum Architekturmodell beziehungsweise Featuremodell zur Komponenten-Bibliothek wird damit nur indirekt modelliert.

4.3.1 Bindung von Komponenten an Parts

Die Strukturfragmente des Architekturmodells beschreiben, welche Struktur die Links zwischen Komponenteninstanzen in der Produktarchitektur haben sollen. Durch die Bindung von Komponenten an Parts der Strukturfragmente wird sichergestellt, dass die geforderte Struktur in der Produktarchitektur vorhanden ist.
Grundsätzlich lässt sich jede Komponente an jeden beliebigen Part binden. Im Kontext des Strukturfragments muss dabei die Randbedingung eingehalten werden, dass die durch die Konnektoren geforderten Links zwischen den Komponenten hergestellt werden können. Das heißt, dass die entsprechenden Komponenten zusammen über ein komplementäres Portpaar gleichen Typs verfügen müssen. Diese Bedingung gilt dabei nicht nur für jeden Konnektor einzeln, sondern für alle Konnektoren des Strukturfragments gleichzeitig. Die Menge der Ports, über die die geforderten Interaktionen realisiert werden können, müssen in derselben Konfiguration gekapselt sein. Nur eine Konfiguration einer Komponente wird in der Produktarchitektur nutzbar sein. Alle Ports, die diese nicht referenziert, stehen nicht zur Verfügung. Hierdurch wird die Menge der möglichen Komponentenbindungen eingeschränkt.
Im Kontext des ganzen Architekturmodells wird diese Menge noch weiter begrenzt, da durch die Verschmelzungen von Strukturfragmenten weitere Konnektoren berücksichtigt werden müssen. Die Menge der möglichen

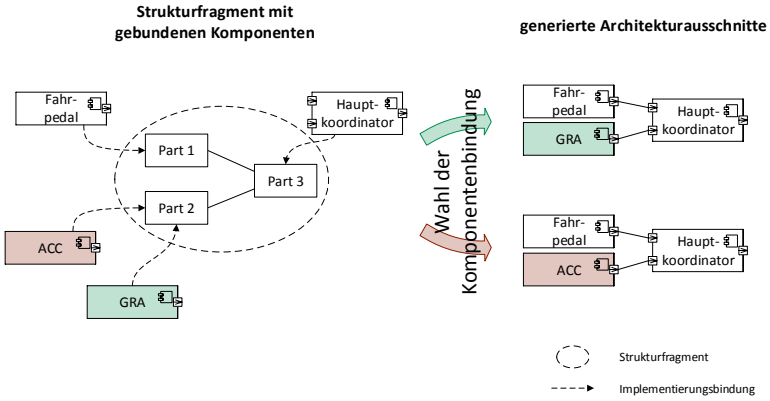


Abbildung 4.12: Komponenten-Variabilität durch alternative Implementierungsbindungen

Komponentenbindungen kann zusätzlich dadurch eingeschränkt werden, dass explizit Porttypen für die Erstellung der Links gefordert werden.
Durch eine Bindung jeweils einer Komponente an jeden Part eines Strukturfragments beschreibt man die Implementierung einer Teilfunktion.

Komponenten-Variabilität durch alternative Implementierungs-Bindungen

Trotz der Einschränkungen hinsichtlich der Bindungen von Komponenten an Parts ist es möglich, dass es mehrere Möglichkeiten für eine Bindung gibt. Diesen Aspekt nutzt man aus, um die Komponenten-Variabilität im *solution space* zu modellieren. An einen Part werden mehrere Komponenten gebunden. Diese Art der Bindung wird im Folgenden Implementierungsbindung genannt. Eine Komponente kann grundsätzlich auch an verschiedene Parts innerhalb eines Fragments gebunden sein. Bei der Erstellung der konkreten Architektur eines Systems, wird sich für eine dieser Implementierungsbindungen entschieden.
So können unter Nutzung eines Strukturfragments verschiedene Ausschnitte einer Softwarearchitektur durch Kombination mit unterschiedlichen Mengen von Implementierungsbindungen beschrieben werden. Abbildung 4.12 zeigt das exemplarisch am Beispiel der Assistenzsystem-Integration. Sie

können wiederkehrende Beziehungsstruktur-Ausschnitte sowohl innerhalb der Architektur eines Systems beschreiben als auch bei Architekturen verschiedener Systeme der Softwareproduktlinie.
Betrachtet man die Implementierungsbindungen im Kontext des gesamten Architekturmodells, müssen Komponenten an Parts mehrerer Fragmente gebunden sein. Durch die Verschmelzung von Fragmenten überlappen sich Parts. An diese Parts können nur Komponenten gebunden sein, die in beiden Fragmenten gleichzeitig mit derselben Konfiguration eingesetzt werden können. Durch eine geeignete Auswahl von Implementierungsbindungen kann somit ein Produkt erzeugt werden, das ein von den Strukturfragmenten gefordertes Beziehungsgeflecht aufweist.

4.3.2 Bindung von Implementierungen an Features

Jede so erstellte Softwarearchitektur wird ein System erzeugen, das bestimmte Merkmale aufweist. Im Rahmen der Softwareproduktlinie ist es wichtig sicherzustellen, dass die Merkmale der erzeugten Systeme gerade eine valide Feature-Konfiguration widerspiegeln.

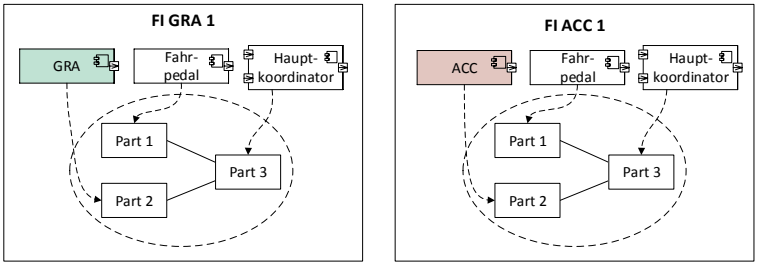


Abbildung 4.13: Darstellung zweier Feature-Implementierungen

Ein Feature wird genau dann durch einen Architekturausschnitt realisiert, wenn eine bestimmte Menge Komponenten in einem bestimmten Beziehungsgeflecht zueinander steht. Eine *Feature-Implementierung* beschreibt genau solch einen Architekturausschnitt. Über die Auswahl einer geeigneten Menge von Implementierungsbindungen wird einer Feature-Implementierung ein Strukturfragment und die benötigte Menge von Komponenten zugeordnet. Eine Feature-Implementierung beschreibt genau eine Variante eines Architekturausschnitts. In Abbildung 4.13 werden zwei

Feature-Implementierungen exemplarisch dargestellt. Die Beziehung zum entsprechenden Feature wird explizit modelliert. In Abbildung 4.14 ist eine Auswahl von Bindungen möglicher Feature-Implementierungen an Features aus dem Beispiel dargestellt[3].
Da ein Feature durch mehrere alternative Implementierungen realisiert werden kann, können mehrere dieser Bindungsmengen mit einem Feature in Verbindung stehen. Im Bild sind diese Feature-Implementierungen rot und grün gefärbt. Umgekehrt können durch eine solche Bindungsmenge mehrere Features realisiert werden. Die grau gefärbte Feature-Implementierung im Bild ist ein Beispiel hierfür.

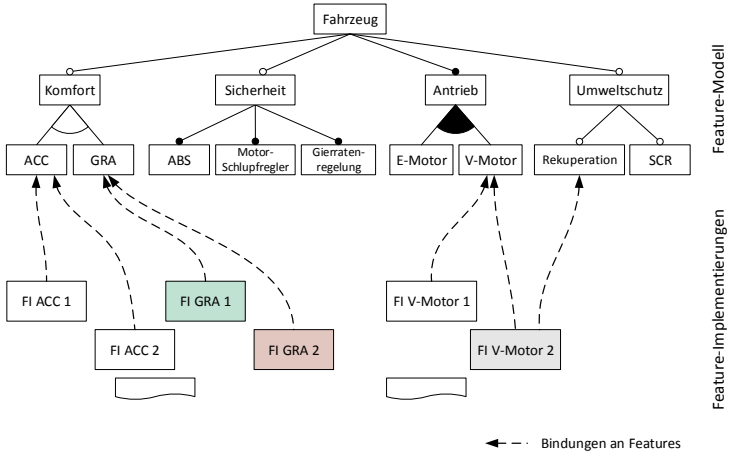


Abbildung 4.14: Feature-Implementierungen werden an Features gebunden

4.4 Ableitung von Architekturen aus dem SPL-Modell

Das Softwareproduktlinienmodell beschreibt auf modulare Weise alle Produkte, die in der Softwareproduktlinie enthalten sind. Das Ziel des *application engineering* ist, aus diesem Modell konkrete Softwarearchitekturen

[3]Die im Modell enthaltenen Querbeziehungen zwischen den Features wurden zugunsten einer besseren Übersichtlichkeit ausgeblendet (vgl. Abbildung 3.3).

mit vorgegebenen Merkmalen abzuleiten (s. Abschnitt 4.1.2). Der folgende Abschnitt zeigt, wie das Softwareproduktlinienmodell für das Erreichen dieses Ziels genutzt wird.
Die Ableitung einer Softwarearchitektur untergliedert sich in sechs Phasen (s. Abbildung 4.15). In jeder Phase wird eine Transformation durchgeführt, die einen Teil der vorhandenen Variabilität eliminiert. Am Ende bleibt genau eine Variante übrig – die Softwarearchitektur des gewünschten Produkts.

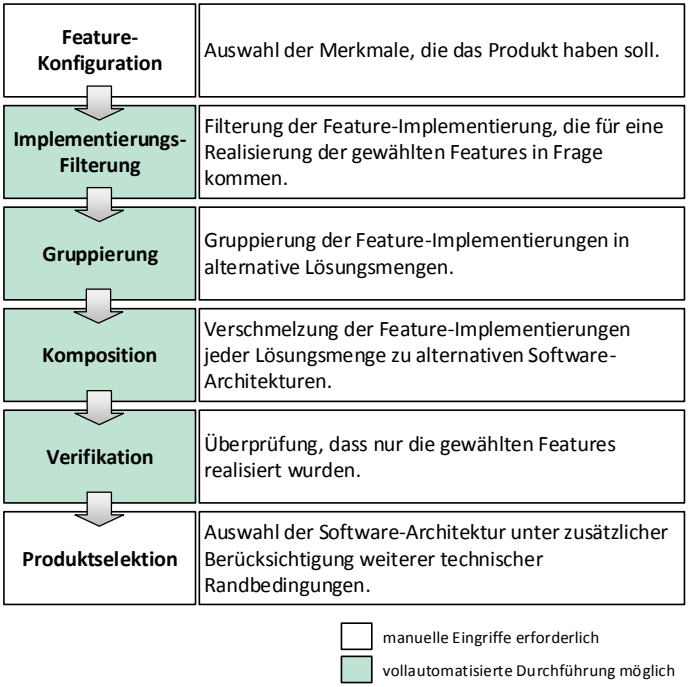


Abbildung 4.15: Die sechs Phasen der Architekturableitung

Bei der Feature-Konfiguration werden die Merkmale, die das Produkt haben soll, ausgewählt. Die Implementierungsfilterung schließt die Feature-Implementierungen von der weiteren Verarbeitung aus, die nicht für die Realisierung der ausgewählten Features in Frage kommen. Zusätzlich wird

die Menge der geeigneten Feature-Implementierungen so zerlegt, dass in jeder Teilmenge keine alternative Feature-Implementierung enthalten ist. Darauf folgend wird versucht, die Elemente dieser Teilmengen zu Softwarearchitekturen zu verschmelzen. Durch die Verifikation wird sichergestellt, dass die einzelnen Softwarearchitekturen nur die geforderten Features realisieren. Bei der abschließenden Produktselektion werden die technischen Randbedingungen berücksichtigt, die direkt durch den Kontext der Domäne einfließen.
In der ersten und letzten Phase muss der Application Engineer manuelle Engriffe machen. Alle anderen Phasen können automatisiert durchgeführt werden.

4.4.1 Feature-Konfiguration

Bei der Spezifikation der Feature-Konfiguration legt der Domänen-Experte fest, welche Merkmale das gewünschte Produkt haben soll. Damit werden die Randbedingungen an das Produkt anhand der fachlichen Aspekte festgelegt (vgl. Abschnitt 3.2.2). Für diesen Schritt wird ausschließlich das Featuremodell genutzt.

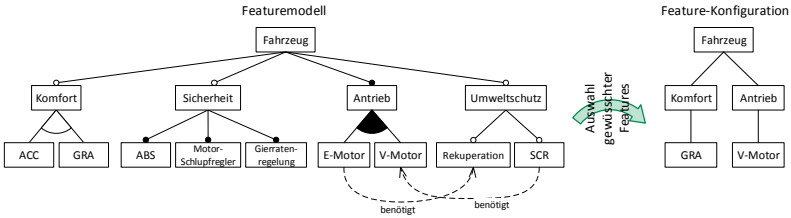


Abbildung 4.16: Beispielhafte Bestimmung einer Feature-Konfiguration

Es wird eine Teilmenge von Features aus dem Featuremodell so ausgewählt, dass keine der im Modell definierten Randbedingungen verletzt wird. Diese Teilmenge definiert die Feature-Konfiguration. In der Feature-Konfiguration ist keine Variabilität bzgl. der Features enthalten. Das bedeutet, dass das zu entwickelnde Produkt alle in dieser Teilmenge enthaltenen Features realisieren muss. Zusätzlich gilt, dass das Produkt keins der Features aus dem Komplement dieser Teilmenge in Bezug auf die Menge der Features aus dem Featuremodell realisieren darf.

In Abbildung 4.16 ist die Konfiguration des Produkts „Mittelklasse“ aus Abschnitt 3.1 (s. Abbildung 3.8) in einer Baumstruktur dargestellt.

4.4.2 Implementierungsfilterung

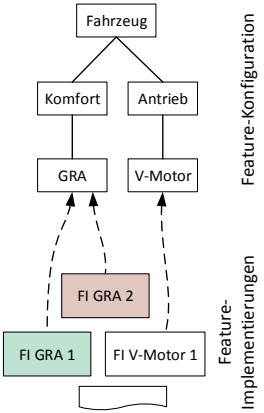


Abbildung 4.17: Für die Architekturerstellung nicht geeignete Feature-Implementierungen werden entfernt

Im Softwareproduktlinienmodell sind Module enthalten, aus denen sich alle Softwarearchitekturen der Produktlinie erstellen lassen. Nicht alle enthaltenen Varianten sind geeignet, um das gewünschte Produkt zu konstruieren. Bei der Implementierungsfilterung werden nicht geeignete Feature-Implementierungen von der Erstellung einer Architektur ausgeschlossen.
Nicht geeignet sind die Feature-Implementierungen, die mit Features in Beziehung stehen, die nicht in der Feature-Konfiguration enthalten sind. In Abbildung 4.17 wird exemplarisch gezeigt, wie das Modell nach der Feature-Konfiguration aussieht. Beim Vergleich mit der Abbildung 4.14 ist zum Beispiel erkennbar, dass die Feature-Implementierung *FI ACC 1* nicht mehr vorhanden ist. Auch die Feature-Implementierung *FI V-Motor 2* wird nach der Filterung nicht mehr genutzt. Obwohl sie das in der Feature-Konfiguration enthaltene Feature *VMotor* realisiert, ist diese Feature-Implementierung nicht geeignet. Sie würde gleichzeitig das

Feature *Rekuperation* realisieren, das jedoch in der Feature-Konfiguration ausgeschlossen wurde.

4.4.3 Gruppierung der Feature-Implementierungen

Ein Feature kann grundsätzlich durch mehrere alternative Feature-Implementierungen realisiert werden. In Abbildung 4.17 repräsentieren die eingefärbten Feature-Implementierungen ein Beispiel für eine Alternative. In einem Produkt soll genau eine Realisierungsalternative für jedes Feature vorkommen. An dieser Stelle in der Entwicklung ist nicht bekannt, welche Alternative die am besten geeignete für das Produkt ist. Darum müssen alle Variationen, die die gewünschten Features realisieren können, berücksichtigt werden.
Die Folge daraus ist, dass mehrere unterschiedliche Mengen von Feature-Implementierungen einer Feature-Konfiguration zugeordnet werden können. Diese unterschiedlichen Mengen definieren die Konfigurationsvarianten. In Abbildung 4.18 werden exemplarisch zwei Konfigurationsvarianten dargestellt, die aus der Menge der in Abbildung 4.17 gezeigten Feature-Implementierungen abgeleitet werden.

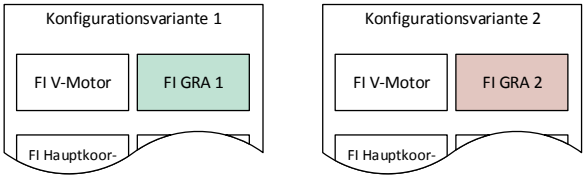


Abbildung 4.18: Potentielle Lösungsmengen von Feature-Implementierungen werden gruppiert

4.4.4 Produktkomposition

In der Phase der Produktkomposition werden alle Feature-Implementierungen, die einer Konfigurationsvariante zugeordnet sind, zu einem System zusammengefügt. Dabei wird die in Abschnitt 4.2.2 vorgestellte Technik der Strukturfragment-Verschmelzung eingesetzt. Abbildung 4.19 zeigt an einem Beispiel, wie zwei Feature-Implementierungen verknüpft werden.

Zur Erinnerung: Eine Feature-Implementierung beschreibt, welche Komponenten an welche Parts eines Strukturfragments gebunden sind. Um Strukturfragmente zu verschmelzen, können grundsätzlich beliebig viele Parts verschmolzen werden, auch bereits verschmolzene Parts.
Zusätzlich zu den im Abschnitt *Strukturfragmente als Graph* genannten Bedingungen bezüglich der Umgebungsparts verringern weitere Regeln die Menge der möglichen Verschmelzungslösungen. Für diese Regeln werden die Bindungen der Komponenten an Parts wie auch die in einer Komponentenkonfiguration verfügbaren Ports referenziert. In der folgenden Auflistung werden die wesentlichen Bedingungen bezüglich dieser Bindungen für eine Verschmelzung gezeigt[4].

- Die Parts müssen Teil verschiedener Strukturfragmente sein.
- An die zu verschmelzenden Parts muss dieselbe Komponente gebunden sein.
- Dieselbe Komponentenkonfiguration muss gewählt sein.
- Bei einer gewählten Konfiguration der Komponente muss eine Menge Ports verfügbar sein, die an die Konnektorenden der verschmolzenen Fragmente gebunden sind.
- Für jede gewählte Portbindung an einem Konnektorende muss ein komplementärer Port des gleichen Porttyps am anderen Konnektorende gewählt sein.

Nicht alle Mengen von Feature-Implementierungen werden diese Randbedingungen erfüllen. Die entsprechenden Mengen dürfen für die weitere Verarbeitung nicht mehr als Konfigurationsvariante zur Verfügung stehen. Bei anderen Mengen wird es mehrere Lösungen geben, die alle Randbedingungen einhalten. In diesem Fall werden alle Architekturbeschreibungen weiterhin berücksichtigt. Als Ergebnis liegen nach dieser Phase mehrere Softwarearchitekturen vor.

4.4.5 Verifikation

Durch die Verschmelzung mehrerer Feature-Implementierungen können Strukturen zufällig entstehen, die ungewünschte Features realisieren. Diese Emergenz lässt sich nicht grundsätzlich vermeiden. Einige solcher Softwarearchitekturen können jedoch automatisch identifiziert und verworfen

[4] Weitere Bedingungen werden im Abschnitt 5 definiert.

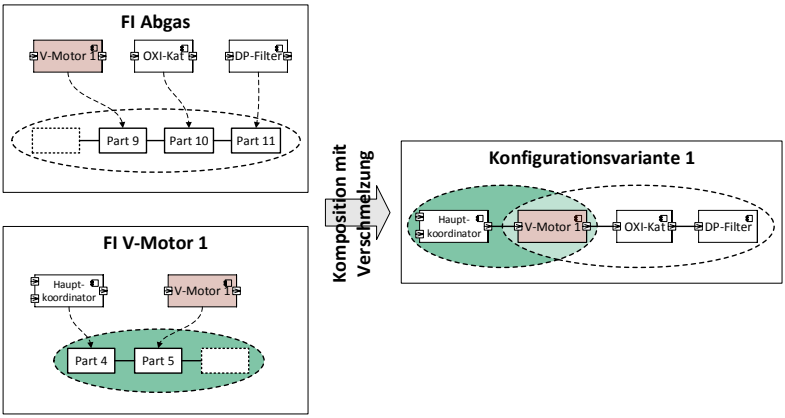


Abbildung 4.19: Die Feature-Implementierungen einer Gruppe werden zu einer Architektur zusammengefügt

werden. In der Verifikations-Phase der Architektur-Ableitung muss geprüft werden, ob die verschmolzenen Architekturen genau die ausgewählten Features realisieren.
Dazu werden die bereits in der Implementierungsfilterung verworfenen Feature-Implementierungen herangezogen. In allen erstellten Architekturen werden Teilstrukturen gesucht, die isomorph zu den ungeeigneten Feature-Implementierungen sind. Existiert eine solche Teilstruktur, bedeutet das, dass das entsprechende Produkt ein Feature realisiert, das es nicht haben soll. Die Softwarearchitektur wird dann verworfen. Das Ergebnis dieser Phase ist eine Menge von Softwarearchitekturen, von denen keine ungewünschte Features realisiert.

4.4.6 Architekturselektion

Bei der Ableitung der Softwarearchitekturen bis einschließlich der Verifikations-Phase wurden nur die fachlichen Aspekte aus dem Featuremodell berücksichtigt. Die Randbedingungen, die durch die technischen Aspekte des Kontextes in der Domäne entstehen, dürfen bei der endgültigen Wahl einer Softwarearchitektur nicht vernachlässigt werden.
Das entwickelte Modellierungskonzept sieht nicht vor, dass die dafür benötigten Informationen im Softwareproduktlinienmodell enthalten sind.

Darum ist es unmöglich, die Entscheidung, welche Architektur auch die technischen Randbedingungen erfüllt, automatisiert zu treffen. Diese Aufgabe muss der Application Engineer manuell erledigen. Er selektiert die geeignete Variante aus der Menge der ausgegebenen Softwarearchitekturen.

4.5 Zusammenfassung

In diesem Kapitel wurde die Idee eines Modellierungskonzepts vorgestellt, mit dem die Ziele der Arbeit erreicht werden können. Die Lösung basiert auf der Kernidee, die verschiedenen Variabilitätsformen explizit und unabhängig voneinander modular zu modellieren.
Die Beschreibung beginnt beim Entwicklungsprozess, der einzelne zum Teil parallel ausführbare Schritte definiert. Für die Darstellung der Systemstruktur wird das Beziehungsgeflecht von konkreten Komponenten abstrahiert und in Strukturfragmente zerlegt. Das Besondere an Strukturfragmenten ist, dass sie nicht disjunkte Ausschnitte beschreiben, die zusammengesteckt werden. Vielmehr wird die vorgesehene Überlappung der Strukturfragmente genutzt, um sie miteinander zu verschmelzen. Variabilität auf Systemstrukturebene wird durch alternative Verschmelzungslösungen von Strukturfragmenten erreicht. Konfigurationsvariabilität wird durch die Zuordnung mehrerer Konfigurationsbeschreibungen zu einer Komponente realisiert. Für Komponentenvariabilität besteht die Möglichkeit, alternative Komponenten in dem System zu binden.
Durch eine geeignete Bindung der Software-Artefakte untereinander sowie an ein Featuremodell wird ein Softwareproduktlinienmodell modular konstruiert. Aus diesem können alle Produkt-Architekturen abgeleitet werden. Das Kapitel endet mit der Beschreibung eines geeigneten Ableitungsverfahrens.

Kapitel 5

Formalisierung des Konzeptes

Im vorherigen Abschnitt wurde das Konzept zur Modellierung einer Softwareproduktlinie vorgestellt. Dafür wurden einerseits verschiedene Teilkonzepte skizziert, die zur Modellierung der verschiedenen Variabilitätsformen dienen, wie zum Beispiel Strukturfragmente. Andererseits wurde eine Methode vorgestellt, die die Ableitung konkreter Architekturen beschreibt.

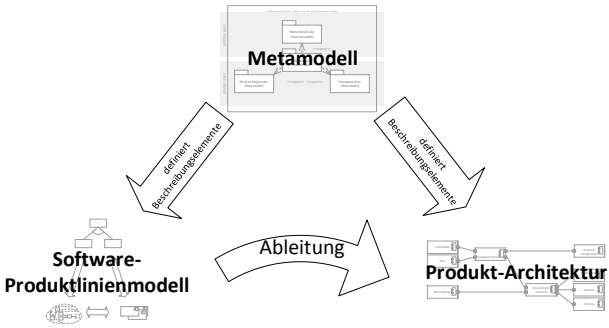


Abbildung 5.1: Darstellung der Zusammenhänge zwischen Modellen und dem Metamodell

Der folgende Abschnitt beschreibt das Konzept auf formale Weise. Dazu wird zuerst das Metamodell spezifiziert, das definiert, wie die Modelle aufgebaut sind (vgl. Kapitel 2). Im zweiten Teil wird dieses Metamodell ergänzt, um zu beschreiben, wie konkrete Produktarchitekturen aus einem entsprechenden Softwareproduktlinienmodell abgeleitet werden können. Abbildung 5.1 zeigt die Zusammenhänge der folgenden Beschreibungen anschaulich. In den ersten beiden Teilen wird die Formalisierung mit den Sprachen UML und OCL durchgeführt. Ein wesentlicher Aspekt bei der

Verbindung zwischen *problem space* und *solution space* wird algorithmisch spezifiziert. Die Beschreibung dieser Lösung wird im letzten Teil dieses Kapitels gezeigt.

5.1 Spezifikation des SPL-Metamodells

Das Metamodell besteht aus vier Teilen (s. Abbildung 5.2). Der erste Teil beschreibt den Aufbau eines Featuremodells. Im zweiten Teil wird spezifiziert, wie Komponenten beschrieben werden können und als drittes wird das Architekturmodell durch das Metamodell für Strukturfragmente definiert. Diese drei Teile werden vom vierten Teil – dem Bindungsmetamodell – importiert und zum Metamodell für Software-Produklinienmodelle vereinigt.
Auf Basis dieser vier Teile ist der Abschnitt weiter untergliedert. Die Unterabschnitte sind dabei immer gleich aufgebaut. Der erste Teil beschreibt die in dieser Dissertation vorgesehene konkrete Syntax. Es wird dabei exemplarisch gezeigt, wie die jeweiligen Artefakte repräsentiert werden können. Für die Spezifikation der abstrakten Syntax kommen UML-Klassendiagramme (s. [Obj12c], [Obj12b]) zum Einsatz. Zusätzliche Modelleigenschaften werden abschließend durch OCL-Ausdrücke definiert (s. [Obj12a]).

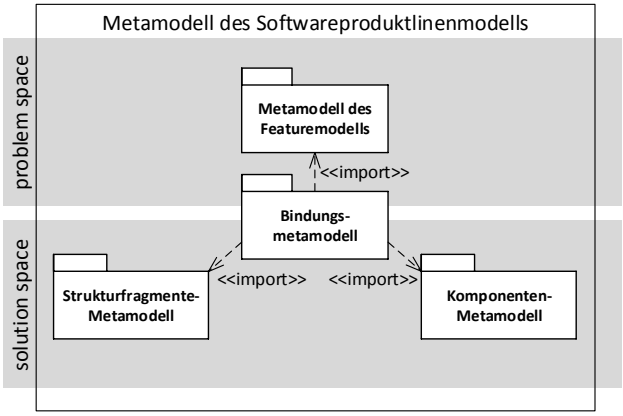


Abbildung 5.2: Bestandteile des Softwareproduktlinien-Metamodells

5.1.1 Spezifikation eines Meta-Featuremodells

Das Featuremodell beschreibt die Variabilität im *problem space*. Im Entwicklungsprozess spielt es insofern eine wichtige Rolle, weil der erste Schritt im *application engineering* die Selektion einer gültigen Feature-Konfiguration ist (s. Abschnitt 4.1). Die inhärenten Regeln, die eine Konfiguration erfüllen muss, um gültig zu sein, spielen für die Ableitung der Produktarchitektur eine untergeordnete Rolle. Für den Ansatz muss das Featuremodell nur wenige Anforderungen erfüllen. Im Grunde reicht die Existenz von referenzierbaren Features aus. Aus diesem Grund ist das im Folgenden vorgestellte Metamodell vergleichsweise einfach.

Konkrete Syntax eines Featuremodells

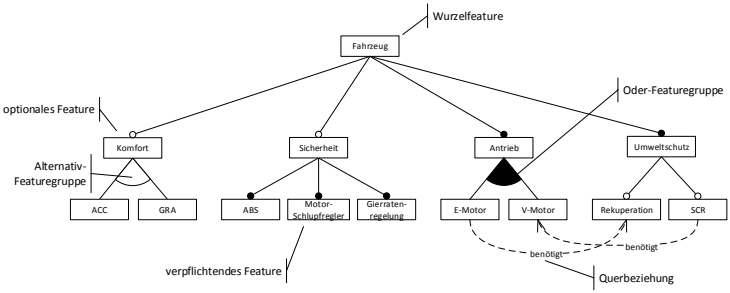


Abbildung 5.3: Konkrete Syntax eines Featuremodells

Das Featuremodell, das in Abbildung 5.3 dargestellt ist, wurde in dieser Arbeit schon mehrfach gezeigt. Daher ist es bestens geeignet, die konkrete Syntax eines Featuremodells zu demonstrieren.
Die Grundstruktur der Featuremodell-Darstellung ist eine Baumstruktur. Features bilden die Knoten, und die hierarchischen Beziehungen werden durch Kanten repräsentiert. Sind Features optional, wird an der Oberkante ein offener Kreis gezeichnet. Bei verpflichtenden Features ist der Kreis ausgefüllt.
Die *Kinder* eines Features können eine Feature-Gruppe bilden, die die Variabilität dieser gruppierten Features genauer spezifiziert. Die Kanten, die Features einer Oder-Gruppe mit ihrem Elternknoten verbinden, werden

durch einen ausgefüllten Kreissektor verbunden. Bei einer Alternativ-Gruppe verbindet ein Kreisbogen die entsprechenden Kanten.
Querbeziehungen sind gerichtete Kanten, die beliebige Features im Modell miteinander verbinden. Zur Darstellung dient ein Pfeil mit einer gestrichelten Linie. Die Art der Querbeziehung (benötigt oder verbietet) wird textuell am Pfeil annotiert.
Da Querbeziehungen jeweils zwei beliebige Features des Modells miteinander in Beziehung setzen, ist der Graph formal kein Baum. Es ist jedoch üblich, die Diagramme aufgrund der hierarchischen Beziehung der Features in einer Baumstruktur darzustellen. Aus diesem Grund werden für die folgenden Beschreibungen Begriffe wie zum Beispiel „Wurzel“ genutzt.

Abstrakte Syntax eines Featuremodells

Die Darstellung eines Featuremodell erfolgt in dieser Dissertation wie gesagt immer in einer Baumstruktur. Für das Modell bedeutet das, dass es genau ein ausgezeichnetes Feature gibt, das die Wurzel bildet. Die Assoziation zwischen den Klassen `Featuremodell` und dem `Feature`, das diese Rolle erfüllt, beschreibt daher eine 1-zu-1-Beziehung.
Alle Features des Featuremodells müssen von den Bearbeitern unterscheidbar sein. Die Möglichkeit, die dafür genutzt werden soll, ist ein eineindeutiger Name jedes Features im Featuremodell. Das Attribut `Name` in der Klasse `Feature` hat deswegen die Eigenschaft `{unique}`. Die Optionalitätseigenschaft eines Features wird über das Attribut `optional` bestimmt.
Die Hierarchisierung der Features wird durch eine reflexive Assoziation realisiert. Dabei kann ein Feature beliebig viele Kinder haben. Bis auf das Wurzel-Feature haben alle Features genau eine Mutter. Das Besondere an dieser reflexiven Assoziation ist, dass sie eine ternäre Assoziation ist, deren drittes Ende mit der Klasse `FeatureGruppe` verbunden ist. Mehrere „Mutter-Kind“-Beziehungen lassen sich damit gruppieren. Die Art der Gruppe (Oder-Feature-Gruppe oder Alternativ-Feature-Gruppe) wird durch das Attribut `gruppenArt` spezifiziert. Da das Featuremodell nur zwei verschiedene Arten von Feature-Gruppen unterstützt, reicht als Attributstyp ein `Boolean` aus. Ein Feature gehört entweder zu genau einer Feature-Gruppe oder ist gar nicht gruppiert.
Querbeziehungen werden durch die gleichlautende Klasse spezifiziert. Diese ist direkt dem Featuremodell zugeordnet. Eine Querbeziehung verbindet immer genau zwei Features. Da sie gerichtet sein kann, müssen die Enden unterschieden werden. Diese Möglichkeit wird durch zwei unterschiedliche Assoziationen zur Klasse `Feature` realisiert. Die unterschiedlichen Arten von Querbeziehungen (benötigt und verbietet) werden durch das Attribut

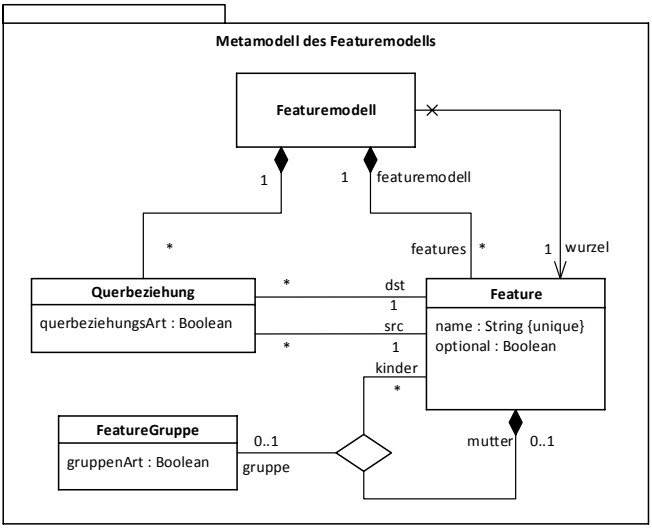


Abbildung 5.4: Abstrakte Syntax eines Featuremodells

`querbeziehungsArt` bestimmt. Da das Featuremodell nur zwei verschiedene Querbeziehungsarten unterstützt, reicht als Attributstyp ein `Boolean` aus.

Randbedingungen in OCL

Die folgenden OCL-Ausdrücke spezifizieren zusätzliche Randbedingungen, die im Modell eingehalten werden müssen. Die Beziehungen zwischen Features eines Featuremodells sollen genau eine Baumstruktur bilden. Die Konsequenz aus dieser Forderung ist, dass es im Modell nur ein Feature geben darf, das keine Mutter hat: das Wurzel-Feature des Featuremodells. Der OCL-Ausdruck 5.1 beschreibt diese Eigenschaft.

```
context Feature
inv: self.mutter = null implies self.featuremodell.wurzel =
    self
```

OCL-Ausdruck 5.1: Das Featuremodell hat nur eine Wurzel

Eine andere Eigenschaft des Featuremodells ist, dass nur *Geschwister-Features*, also Features, die dieselbe Mutter haben, in derselben Feature-Gruppe enthalten sein können. Die ternäre Assoziation garantiert das nicht. Die entsprechende Bedingung ist im OCL-Ausdruck 5.2 definiert.

```
context FeatureGruppe
inv: self.kinder->forAll(f1,f2 : Feature | f1.mutter = f2.
    mutter)
```

OCL-Ausdruck 5.2: Gruppierung nur für Geschwisterfeatures

5.1.2 Spezifikation der Komponenten-Bibliothek

Der folgende Abschnitt beschreibt, aus welchen Elementen die Komponenten-Bibliothek aufgebaut ist. Dieser Teil ist in der Abbildung 5.2 durch das Paket *Komponenten-Metamodell* dargestellt.
Die Komponenten-Bibliothek fasst alle Komponenten, die in der Softwareproduktlinie eingesetzt werden, in einer Menge zusammen. Den wesentlichen Anteil an dem Metamodell macht die Beschreibung des Komponenten-Modells aus. Die Bibliothek repräsentiert „nur“ einen Container, in dem die Komponenten gesammelt werden. Für die Bibliothek ist keine eigene Darstellung vorgesehen. Darum werden im Folgenden nur Beispiele für die Darstellung einzelner Komponenten vorgestellt.

Konkrete Syntax einer Komponente

Die Darstellungen von Komponenten, die in den vorherigen Abschnitten gezeigt wurden (s. zum Beispiel Abbildung 4.11), nutzen immer eine UML konforme Notation. In der Produktlinie aus Abschnitt 3.1 erscheint zum Beispiel die Komponente *Bremskraftverteiler* in unterschiedlichen Formen. Dieser Unterschied ist ein Beispiel für die Konfigurationsvariabilität. Um verschiedene Konfigurationen in einem grafischen Objekt beschreiben zu können, muss die Darstellungsform erweitert werden. Abbildung 5.6 zeigt an diesem Beispiel die erweiterte konkrete Syntax einer Komponente mit mehreren Konfigurationen.
Die rechteckige Grundform der Komponentendarstellung basiert auf der konkreten Syntax nach UML. Zusätzlich zum Namen und dem UML-Komponentensymbol oben rechts werden die Komponenten-Konfigurationen durch eingebettete Rechtecke dargestellt. Alle so dargestellten Komponenten-Konfigurationen werden in einer Spalte übereinander angeordnet. Die Ports, die einer Konfiguration zugeordnet sind, werden über gerade Linien

Abbildung 5.5: Verschiedene Konfigurationen derselben Komponente nutzen unterschiedliche Anzahl an Ports gleichen Typs

mit dem jeweiligen Konfigurations-Rechteck verbunden. Bei Zuordnung zu mehreren Konfigurationen, werden entsprechend viele Linien gezeichnet.

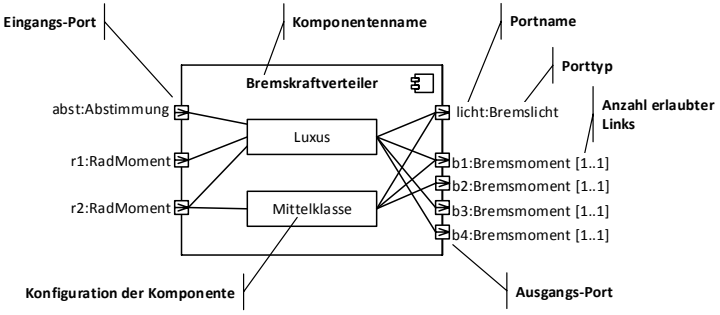


Abbildung 5.6: Die konkrete Syntax am Beispiel einer Komponente mit zwei Konfigurationen

Die Art eines Ports wird durch die Richtung eines Pfeilsymbols unterschieden, die bei Eingangsports in die Komponente hinein und bei Ausgangsports aus ihr heraus zeigt. Weitere Spezifikationen (`<port-name> [':' <port-typ>]['['<linkzahl>']']`) können textuell an dem entsprechenden Port annotiert werden. Der Name des Ports wird durch `<port-name>` repräsentiert. `<port-typ>` ist der Name des assoziierten Port-Typen. Über

`<linkzahl>` kann optional angegeben werden, wieviele Links an dem entsprechenden Port aufgebaut werden können. Wenn keine Angabe hierfür gemacht wird, gilt Eins als erlaubte Anzahl von Links.

Abstrakte Syntax der Komponenten-Bibliothek

In Abbildung 5.7 werden die Elemente des Pakets `Komponenten-Metamodell` dargestellt. Ausgangspunkt für die folgende Betrachtung ist die Klasse `Komponenten-Bibliothek`.
In der Softwareproduktlinie gibt es genau eine Instanz dieser Klasse. Sie hat die Aufgabe, alle Softwarekomponenten der Softwareproduktlinie in einer zentralen Verwaltungs-Einheit zusammenzufassen. Die Kardinalitäten an der Assoziation zwischen den Klassen `Komponenten-Bibliothek` und `Komponente` definieren darum eine 1-zu-*-Beziehung. Da die Komponenten auch außerhalb der Softwareproduktlinie wiederverwendbar sind, ist die Assoziation als Aggregation spezifiziert.
Alle Komponenten werden im Kontext der Bibliothek eindeutig durch ihren Namen identifiziert. Bei der Modellierung einer Implementierungsbindung (s. Abschnitt 4.3.1) dürfen nur Komponenten genutzt werden, die ein spezielles Verhalten aufweisen. Dieses Verhalten wird nicht im Metamodell abgebildet, weshalb eine zusätzliche Information für die Unterscheidung der Komponenten notwendig ist. Die Erstellung der Implementierungsbindung wird durch einen Menschen durchgeführt. Darum wird der Name der Komponente (Attribut: `name:String{unique}`) als Identifikator gewählt, weil darüber eine für Menschen intuitiv durchzuführende Identifikation möglich ist.
Die Menge der Komponenten, die einer Bibliothek zugeordnet sind, umfasst genau alle Komponenten, die im Softwareproduktlinienmodell vorkommen. Darum ist die Angabe der Eigenschaft `{unique}` am Namensattribut bereits hinreichend für die eindeutige Identifizierung.
Im Abschnitt 4.2.3 wird beschrieben, dass die Konfigurations-Variabilität durch verschiedene Konfigurationen einer Komponente realisiert wird. Da Komponenten-Konfigurationen nur im Kontext ihrer Komponente existieren, bildet die Assoziation die Beziehung als Komposition ab. Dabei gilt, dass alle Konfigurationen eindeutig an eine Komponente gekoppelt sind und jede Komponente mindestens eine Konfiguration besitzt. Die verschiedenen Konfigurationen werden über eine Assoziation im Kontext der Komponente identifiziert. Der Konfigurationsname muss nicht eindeutig in der Softwareproduktlinie sein. Aber innerhalb einer Komponente darf kein Konfigurationsname mehrfach vergeben sein. Damit wird eine eineindeutige Identifikation einer Komponenten-Konfiguration im Kontext der SPL

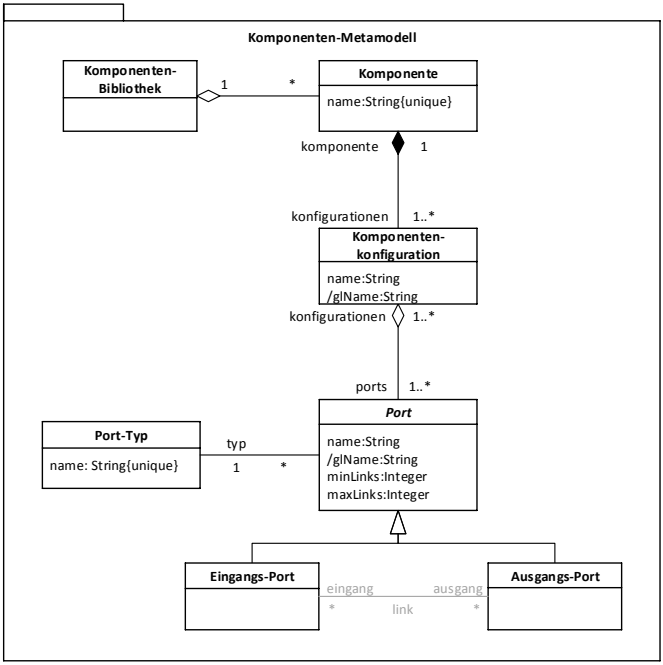


Abbildung 5.7: Abstrakte Syntax zur Beschreibung einer Komponente

unter zusätzlicher Berücksichtigung des Komponentennamen möglich. Der entsprechende Identifikator (`glName:String`) wird abgeleitet. Auch die Konfiguration muss im *domain engineering*-Prozess von einem Menschen ausgewählt werden, was über eindeutige Namen erleichtert wird.

Jede Komponente verfügt über eine Menge Ports, über die die (ggf. verschiedenen) Funktionalitäten der Komponente genutzt werden können. Einer Komponenten-Konfiguration ist eine Teilmenge dieser Ports zugeordnet. Da diese Teilmengen im Kontext einer Komponente nicht disjunkt sind, ist die Assoziation als Aggregation spezifiziert. Ein Port muss in mindestens einer, kann aber in mehreren Konfigurationen genutzt werden. Umgekehrt muss eine Konfiguration mindestens über einen Port verfügen. Die Anzahl hängt von der Funktionalität ab, die zu einer Konfiguration gehört.

Unabhängig von der Anzahl der Portinstanzen, die für eine Konfiguration notwendig sind, ist die Anzahl der Links, die an einem Port aufgebaut werden können. Diese Information ist unabhängig von den Konfigurationen. Die Anzahl der zulässigen Links, die über dem jeweiligen Port aufgebaut werden, wird durch ein Intervall bestimmt. Dabei werden die Grenzen durch die Attribute `minLinks` und `maxLinks` bestimmt.

Die Port-Klasse ist abstrakt. Von dieser Klasse erben die Klassen `Eingangs-Port` und `Ausgangs-Port`. Diese beiden Klassen dienen dazu, die komplementären Eigenschaften bei einer Kommunikation darstellen zu können (zum Beispiel Sender und Empfänger, Dienstanbieter und -nutzer usw.).

Das Port-Paar, über das die Kommunikation realisiert wird, ist im Metamodell durch die Assoziation `link` verbunden. Die Assoziation ist in Abbildung 5.7 aufgehellt dargestellt. Der Link wird erst später in diesem Kapitel spezifiziert. Die Darstellung ist an dieser Stelle stark vereinfacht, denn wie man sehen wird, ist der Link eine ternäre Assoziation. Obwohl die Beschreibung an dieser Stelle noch unvollständig ist, soll die Beziehung bereits genannt werden. Eine wesentliche Eigenschaft eines Links ist in dem gezeigten Ausschnitt gut erkennbar: die Gleichheit der Port-Typen der durch einen Link verbundenen Ports.

Die gemeinsamen Eigenschaften eines komplementären Portpaars werden durch einen Port-Typen spezifiziert. Dabei handelt es sich zum Beispiel um die Information, ob synchron oder asynchron kommuniziert wird, welche Datentypen übertragen werden oder dass ein bestimmtes Protokoll genutzt werden muss. Bei der Ableitung von Architekturen aus dem Softwareproduktlinienmodell wird bezüglich des Typs nur Gleichheit verlangt. Im *domain engineering* werden konkrete Eigenschaften eines Ports auf einen Port-Typen abgebildet und jedem Port genau der entsprechende Port-Typ zugeordnet. Der Port-Typ kann in der Softwareproduktlinie an einem eineindeutigem Namen identifiziert werden, was durch das Schlüsselwort `unique` spezifiziert ist.

Randbedingungen in OCL

Das in Abbildung 5.7 beschriebene Metamodell spezifiziert nicht alle Eigenschaften, die das Modell haben soll. Die folgenden OCL-Ausdrücke ergänzen hierfür entsprechende Randbedingungen.

Alle Ports, die einer Komponenten-Konfiguration zugeordnet sind, gehören genau zu der Komponente, die die Komponenten-Konfiguration beinhaltet. Diese Eigenschaft wird durch den OCL-Ausdruck 5.3 definiert.

```
context Komponente
inv: self.konfigurationen.ports
                  ->forAll(p:Port | p.konfigurationen.
                      komponente = self)
```

OCL-Ausdruck 5.3: Ports der Komponentenkonfiguration gehören zur Komponente

Bezüglich der Kommunikation über Ports gibt es die Einschränkung, dass eine Komponente nicht mit sich selbst über einen Link verbunden sein darf. Obwohl sich die Eigenschaften eines `link` im Rahmen des Pakets `Komponenten-Metamodell` noch nicht vollständig beschreiben lassen, kann diese Kommunikations-Einschränkung mit Hilfe des Links beschrieben werden (s. OCL-Ausdruck 5.16).

```
context Komponente
inv: konfigurationen.ports
                  ->forAll(p:Ausgangs-Port |
                           p.eingang.konfigurationen.
                               komponente <> self)
inv: konfigurationen.ports
                  ->forAll(p:Eingangs-Port |
                           p.ausgang.konfigurationen.
                               komponente <> self)
```

OCL-Ausdruck 5.4: Einschränkungen hinsichtlich der Links

Eine Komponenten-Konfiguration ist im Kontext einer Komponente eindeutig identifizierbar. Durch die Assoziation zur Komponente und die eindeutige Identifizierbarkeit einer Komponente im Kontext der SPL ist auch die Komponenten-Konfiguration im Kontext der SPL eindeutig identifizierbar. Um den Vergleich zweier Konfigurationen zu vereinfachen, wird das Attribut `glName` aus den Identifikatoren der Komponenten-Konfiguration und der dazugehörigen Komponente abgeleitet (s. OCL-Ausdruck 5.5).

```
context Komponenten-Konfiguration::glName
derive: komponente.name.concat('.').concat(name)
```

OCL-Ausdruck 5.5: Ableitung des global eindeutigen Konfigurationsnamen

Eine vergleichbare Forderung lässt sich für den Port spezifizieren. Der OCL-Ausdruck 5.6 definiert die entsprechende Randbedingung.

```
context Port::glName
derive: konfigurationen.komponente.name.concat('.').concat(
    name)
```

OCL-Ausdruck 5.6: Ableitung des global eindeutigen Portnamen

An dieser Stelle sei darauf hingewiesen, dass die OCL-Ausdrücke 5.5 und 5.6 nicht notwendig für eine eindeutige Identifizierung sind. Die Beschreibungen in Abbildung 5.7 reichen dafür aus. Diese zusätzlichen Bedingungen dienen nur der einfacheren Identifizierung durch einen Menschen während des *domain engineering*s.

Das Intervall zur Angabe der erlaubten Anzahl von Links an einem Port wird wie gesagt durch die Attribute `minLinks` und `maxLinks` spezifiziert. Das Minimum aller möglichen unteren Grenzen bildet dabei die Null, denn es ergibt keinen Sinn, weniger als keinen Link zuzulassen. Das sinnvolle Minimum aller möglichen oberen Grenzen bildet Eins. Denn ein Port, über den im Höchstfall keine Interaktion stattfinden darf, widerspricht dem eigentlichen Sinn des Portkonzepts. Dennoch erlaubt die zweite Invariante im OCL-Ausdruck 5.7 auch die Null. Die Semantik bei Belegung mit Null weicht von allen anderen Belegungen ab. In diesem Fall wird nicht das Maximum der möglichen Links an dem entsprechenden Port spezifiziert. Stattdessen ist die Bedeutung, dass keine Begrenzung der Link-Anzahl nach oben gefordert ist. Wenn `maxLinks` mit einem Wert größer als Null belegt ist, gilt, dass der Wert von `minLinks` immer kleiner oder gleich dem von `maxLinks` ist. Auf diese Weise kann das Intervall entweder als ein abgeschlossenes oder rechtsseitig unendliches abgeschlossenes Intervall spezifiziert werden.

```
context Port
inv: minLinks >= 0
inv: maxLinks >= 0
inv: (maxLinks = 0) or (minLinks <= maxLinks)
```

OCL-Ausdruck 5.7: Beschränkung der Link-Kardinalitätsattribute

5.1.3 Spezifikation eines Architekturmodells

Das Architekturmodell beschreibt, welche Strukturen das Beziehungsgeflecht eines Systems im Rahmen der Softwareproduktlinie annehmen kann. Es besteht aus einer Menge von Strukturfragmenten, die auf verschiedene Weise miteinander verschmolzen werden können. Da die Strukturfragmente selbst die Informationen für mögliche Verschmelzungslösungen enthalten, wird im folgenden Primär der Aufbau eines Strukturfragments beschrieben.

Konkrete Syntax eines Strukturfragments

In Abbildung 5.8 sind Ausschnitte von zwei System-Varianten dargestellt, die in Kapitel 3.1 vorgestellt wurden. In diesen Ausschnitten werden unterschiedliche Komponentenmengen genutzt, deren Beziehungsgeflechte aber eine vergleichbare Struktur bilden. Diese Struktur wird im Folgenden genutzt, um die konkrete Syntax eines Strukturfragments zu erklären.

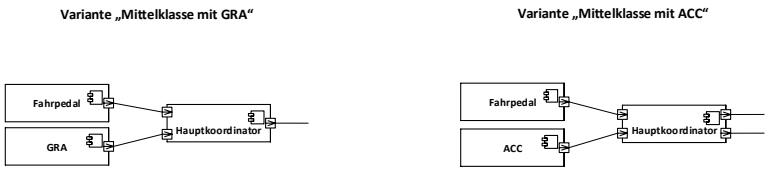


Abbildung 5.8: Ausschnitte zweier Varianten mit vergleichbarem Beziehungsgeflecht

Das Strukturfragment, das das Beziehungsgeflecht aus diesem Ausschnitt beschreiben kann, ist in Abbildung 5.9 dargestellt. Ein Strukturfragment besteht aus einer Menge von Parts. Parts, an die eine Komponente direkt gebunden werden kann, sind als Rechteck mit durchgezogener Linie dargestellt. Der Name des Parts befindet sich innerhalb des Rechtecks. Umgebungsparts dienen der Struktur-Beschreibung der unmittelbaren Umgebung. Für die Darstellung wird ebenfalls ein Rechteck genutzt, allerdings mit gestrichelter Linie. Die Konnektoren werden als durchgezogene Linie zwischen den verbundenen Parts dargestellt.
Um die Grenzen eines Strukturfragments kenntlich zu machen, wird es innerhalb eines Ovals mit gestrichelter Linie dargestellt. An der Oberseite des ovalen Bereichs kann der Name des Strukturfragments angegeben werden.

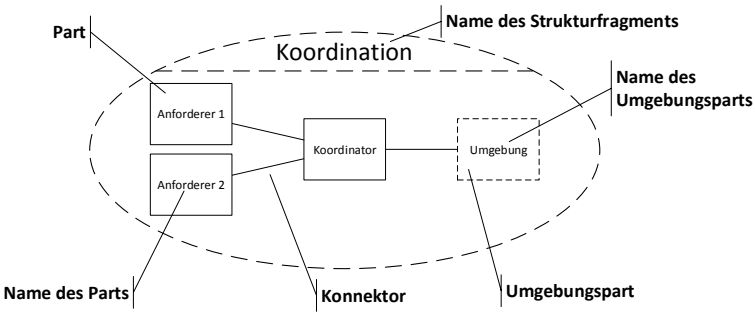


Abbildung 5.9: Konkrete Syntax eines Strukturfragments

Abstrakte Syntax eines Strukturfragments

Der Einstiegspunkt bei der Beschreibung der abstrakten Syntax ist die Klasse `Architekturmodell`. Das Architekturmodell besteht aus allen in der Softwareproduktlinie enthaltenen Strukturfragmenten.
Die Assoziation zwischen den Klassen `Architekturmodell` und `Strukturfragment` ist darum als Komposition spezifiziert, die eine 1-zu-*-Beziehung repräsentiert. Da Strukturfragmente vom Architekten beim Modellieren eindeutig identifizierbar sein müssen, wird der Name als ein Attribut vom Typ `String` mit der Eigenschaft `unique` definiert.
Ein Strukturfragment besteht aus den Elementen Parts, Umgebungsparts und Konnektoren. Die Assoziation zwischen Strukturfragment und Konnektor beschreibt eine 1-zu-*-Komposition. Es können grundsätzlich beliebig viele Konnektoren enthalten sein. Es ist auch möglich, dass im Strukturfragment kein Konnektor existiert, zum Beispiel wenn nur ein Part enthalten ist.
Parts und Umgebungsparts haben gemeinsam, dass sie innerhalb eines Strukturfragments über Konnektoren verbunden werden und dass beide einen Namen besitzen können. Darum werden die gemeinsamen Eigenschaften in einer abstrakten Klasse (`AbsPart`) zusammengefasst von der sowohl die Klasse `Part` als auch `Umgebungspart` erben. Die Sinnhaftigkeit der Aufteilung von Part und Umgebungspart in zwei Klassen wird erst bei der Spezifikation der Bindungen erkennbar. Der wesentliche Unterschied besteht darin, dass nur an Parts Komponenten gebunden werden können. Die Beziehung eines Strukturfragments zu seinen Parts und Umgebungsparts wird über eine Komposition beschrieben. Die Kardinalität auf `parts`-

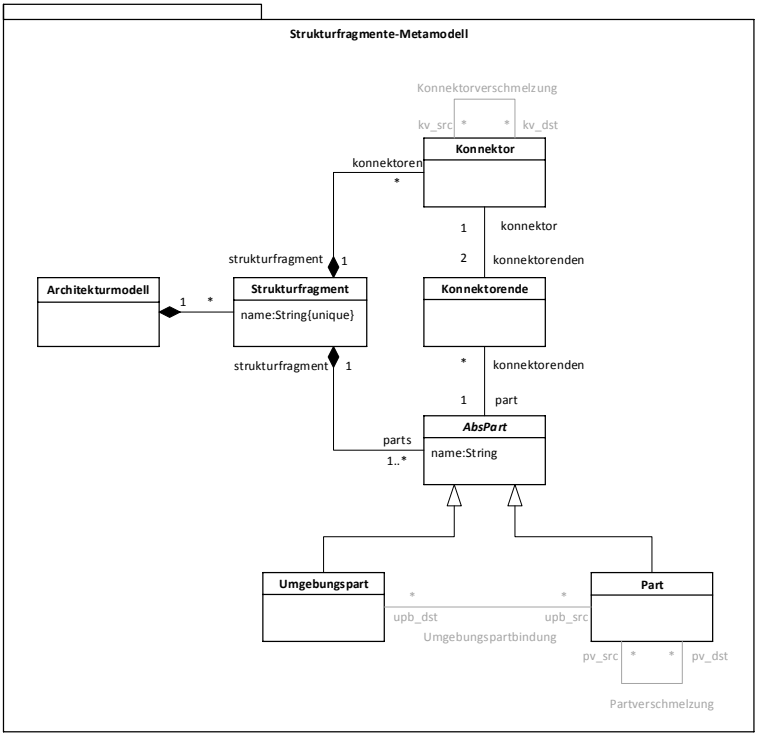


Abbildung 5.10: Abstrakte Syntax eines Strukturfragments

Seite ist `1..*`, da ein Strukturfragment grundsätzlich aus beliebig vielen, mindestens aber aus einem Part besteht.

Ein Konnektor verbindet immer genau zwei *AbsParts*. Um die beiden Enden des Konnektors explizit im Modell adressieren zu können, wird die Assoziationsklasse `Konnektorende` eingeführt. Die Kardinalitäten an der Assoziation im Metamodell zwischen Konnektor und Konnektorende sind 1 bzw. 2, da jedes Konnektorende zu genau einem Konnektor gehört, und dieser genau zwei Enden hat. Die Assoziation zwischen Konnektorende und AbsPart beschreibt eine *-zu-1-Beziehung. Jedes Konnektorende steht jeweils mit genau einem Part oder Umgebungspart in Beziehung. Umge-

kehrt können von einem Part bzw. Umgebungspart grundsätzlich beliebig viele Konnektoren ausgehen.
Die Verschmelzung von Strukturfragmenten wird realisiert durch die Überlagerung der Elemente verschiedener Strukturfragmente. Die Assoziationen `Konnektorverschmelzung` und `Partverschmelzung` beschreiben die Überlagerung von Konnektoren beziehungsweise Parts. Kennzeichnend für diese Verschmelzungen ist, dass die entsprechenden Elemente vom gleichen Typ sind. Die Assoziation `Umgebungspartbindung` repräsentiert die Überlagerung eines Parts mit einem Umgebungspart, also Elemente unterschiedlichen Typs. Alle Elementverschmelzungen und Bindungsassoziationen werden abgeleitet. Das ist allerdings nur im Kontext der Softwareproduktlinie möglich, da bekannt sein muss, welche Komponenten an die Parts gebunden sind. Wie schon bei der Beschreibung der Assoziation `link` im Abschnitt 5.1.2 werden diese Assoziationen aufgehellt dargestellt, da die eigentliche Spezifikation in einem späteren Abschnitt folgt.
Eine Partverschmelzung kann nur zwischen Parts realisiert werden. Die Bedingung hierfür ist, dass dieselbe Komponente an die zu verschmelzenden Parts gebunden ist. Da an Umgebungsparts keine Komponenten gebunden werden können, ist eine Verschmelzung von Umgebungsparts bzw. eines Umgebungsparts mit einem Part nicht möglich. Die Beziehung zwischen Umgebungspart und Part wird Umgebungspartbindung genannt. Der Part repräsentiert dabei den Teil der Umgebungsstruktur, die durch den Umgebungspart gefordert wird (s. Abschnitt 4.2.2).

Randbedingungen in OCL

Auch das Metamodell des Architekturmodells (Abbildung 5.10) beschreibt noch nicht alle Eigenschaften, die das Modell haben soll. Die erste wichtige Randbedingung (s. OCL-Ausdruck 5.8) besagt, dass alle Parts beziehungsweise Umgebungsparts, die über Konnektoren miteinander verbunden sind, zum selben Strukturfragment gehören wie auch die Konnektoren selbst.

```
context Strukturfragment
inv: self.parts->forAll(p:AbsPart | p.konnektorenden.
 konnektor->forAll(k:Konnektor | k.strukturfragment = self))
inv: self.konnektoren->forAll(k:Konnektor| k.konnektorenden.
 part->forAll(p:Part | p.strukturfragment = self))
```

OCL-Ausdruck 5.8: Part-Konnektor-Geflecht gehört zu einem Strukturfragment

Wichtig bei der Verbindung durch einen Konnektor ist, dass ein Part oder Umgebungspart nicht mit sich selbst in Beziehung stehen darf. Die beiden Enden des Konnektors sind entsprechend mit verschiedenen Elementen verbunden.

```
context Konnektor
inv: self.konnektorende
        ->forAll(e1, e2 | e1 <> e2 implies e1.part <> e2.
            part)
```

OCL-Ausdruck 5.9: Konnektoren sind nicht reflexiv

5.1.4 Spezifikation der Bindungen im Produktlinienmodell

Der letzte Schritt im *domain engineering* ist die Bindung von Features, Strukturfragmenten und Komponenten (s. Abschnitt 4.1). Der in diesem Schritt zu erstellende Teil des Softwareproduktlinienmodells spielt eine zentrale Rolle, um im *application engineering* eine automatisierte Ableitung durchführen zu können. In Abbildung 4.3 ist dargestellt, dass das Bindungs-Metamodell die zuvor beschriebenen Metamodellteile zum Metamodell des Softwareproduktlinienmodells zusammenfügt.

Konkrete Syntax der Bindungen

Bei der Darstellung der Bindungen im Softwareproduktlinienmodell muss eine besondere Herausforderung gemeistert werden. Würde eine einzige Darstellungsform spezifiziert werden, müssten gleichzeitig Elemente aus allen Modellteilen sowie deren Beziehungen vollständig dargestellt werden. Die Komplexität des Dargestellten wäre sehr schnell unüberschaubar für die Architekten. Aus diesem Grund werden mehrere Sichten aus insgesamt vier unterschiedlichen Blickwinkeln spezifiziert, die sich auf die Darstellung der jeweils wesentlichen Eigenschaften für eine bestimmte Aufgabe im Prozess beschränken.
Mit der *Modellierungssicht* wird genau eine Feature-Implementierung (s. auch Abschnitt 4.3.2) dargestellt. Diese Sicht ist gut geeignet, die Bindungen im letzten *domain engineering*-Schritt herzustellen. Die Aufgabe des Architekten in diesem Schritt ist festzulegen, welche Zuordnung Komponentenkonfigurationen zu Parts bzw. Ports zu Konnektoren geeignet ist, um bestimmte Features zu realisieren. Das bedeutet, dass zum Beispiel andere Komponenten, die in anderen Featureimplementierungen zum Einsatz kommen, ausgeblendet sind.

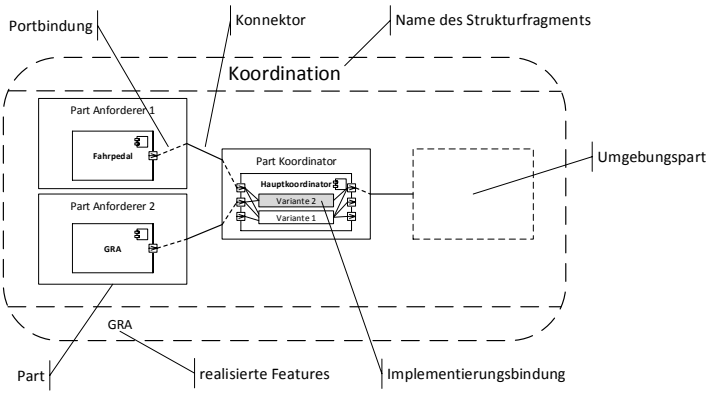


Abbildung 5.11: Beispiel einer Featureimplementierung in der Modellierungssicht

Diese Sicht beschreibt einen Blickwinkel, der – bezogen auf die in Abbildung 5.2 dargestellten Struktur – von *innen* nach *außen* gerichtet ist. Die Darstellung wird exemplarisch in Abbildung 5.11 gezeigt.
Da eine Feature-Implementierung immer genau ein Strukturfragment nutzt, wird die Grundstruktur der Darstellung durch das Strukturfragment gebildet. Der Unterschied zur Strukturfragmentdarstellung (s. Abbildung 5.9) ist die andere Form des Rahmens. Die Grundform des Rahmens ist bei einer Implementierungsbindung ein Rechteck mit abgerundeten Ecken. An der oberen Seite ist ein Bereich für die Bezeichnung des Strukturfragments vorgesehen. An der unteren Seite werden die Features angegeben, die durch die Implementierungsbindung realisiert werden.
Zur Darstellung der Implementierungsbindung wird die gebundene Komponente innerhalb des entsprechenden Partrechtecks gezeichnet. Verfügt die Komponente über mehrere Konfigurationen, wird die gebundene farblich markiert. Portbindungen werden durch gestrichelte Linien repräsentiert, die den jeweiligen Port mit dem entsprechenden Konnektorende verbindet.

In der *Reportsicht* wird im Gegensatz zur Modellierungssicht bezogen auf die Metamodellstruktur von *außen* nach *innen* geblickt. Da es drei importierte Metamodellteile gibt, wird die Reportsicht in weitere drei Blickwinkel unterteilt. Mit dieser Sicht lässt sich die Variabilität im Softwareproduktlinienmodell gut darstellen. Beispielsweise kann mit diesen

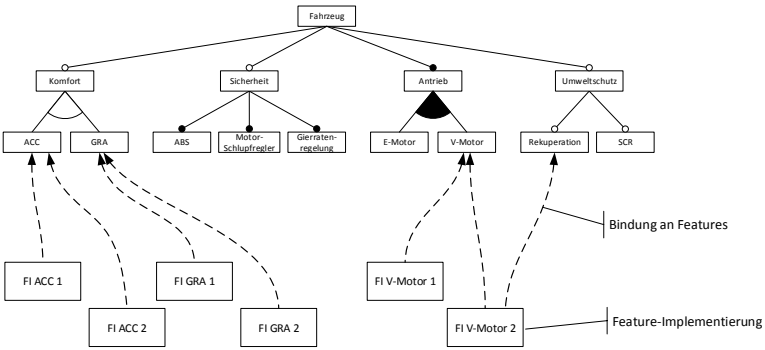


Abbildung 5.12: Reportsicht aus Blickwinkel des Featuremodells

Darstellungsformen die Auswahl verschiedener Feature-Implementierungen bei Erstellung einer Feature-Konfiguration gut visualisiert werden. Die verschiedenen Darstellungsmöglichkeiten wurden bereits in Abschnitt 4.3 angewendet.

Abbildung 5.12 zeigt die Reportsicht aus dem Blickwinkel des Featuremodells. Feature-Implementierungen werden als Rechtecke dargestellt. Die Beziehung zu den Features, die sie jeweils realisieren, wird über gestrichelte Pfeile dargestellt. Dabei werden nur die Implementierungen dargestellt, die an ausgewählte Features geknüpft sind. Die große Anzahl an Features und Feature-Implementierungen würde diese Darstellung sonst sehr schnell unüberschaubar machen.

Der zweite Blickwinkel in der Reportsichtweise wird in Abbildung 5.13 dargestellt. Da in jeder Feature-Implementierung prinzipiell immer nur ein Strukturfragment eingesetzt wird, ist es sinnvoll, mehr Details der Implementierung zu zeigen. Die interessante Information ist in diesem Fall, welche Komponenten in den verschiedenen Featureimplementierungen an die entsprechenden Parts gebunden sind. Die Implementierungsbindungen werden als gestrichelte Pfeile dargestellt. Dabei werden nur die Komponentenkonfigurationen dargestellt, die an den entsprechenden Part gebunden sind. Werden unterschiedliche Komponentenkonfigurationen derselben Komponente an verschiedene Parts gebunden, wird die Komponente mehrfach gezeichnet, jeweils mit der entsprechenden Konfiguration. Die realisierten Features werden in Form einer Liste textuell dargestellt.

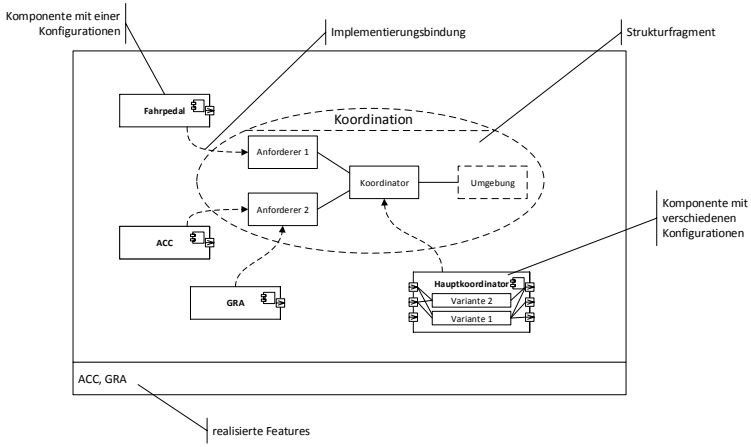


Abbildung 5.13: Reportsicht aus dem Blickwinkel eines Strukturfragments

Die letzte Reportsicht beschreibt den Blick aus Richtung der Komponente. In Abbildung 5.14 ist exemplarisch gezeigt, in welchen Featureimplementierungen eine Komponente eingesetzt werden kann.

Abstrakte Syntax der Bindungen

Die verschiedenen Sichten beschreiben alle Teile des Modells, in denen die Bindungen von zentraler Bedeutung sind. Die abstrakte Syntax des Bindungsmetamodells ist in Abbildung 5.15 dargestellt.
Der Ausgangspunkt bei der Beschreibung der abstrakten Syntax bildet die Klasse `Feature`. In Abschnitt 4.3.2 wurde beschrieben, dass ein Feature durch mehrere alternative Implementierungen realisiert werden kann. Die Assoziation zwischen Feature und Feature-Implementierung definiert darum eine 1-zu-*-Beziehung.
Eine Feature-Implementierung besteht zum Einen aus der Beschreibung, welche Komponenten an welche Parts eines Strukturfragments gebunden sein müssen, damit ein Feature realisiert wird. Dabei muss sichergestellt sein, dass die korrekte Konfiguration gewählt ist. Die Zuordnung von Komponentenkonfigurationen zu Parts wird durch die Klasse `Implementierungsbindung` realisiert. Jede konkrete Implementierungsbindung existiert nur im Kontext der jeweiligen Feature-Implementierung.

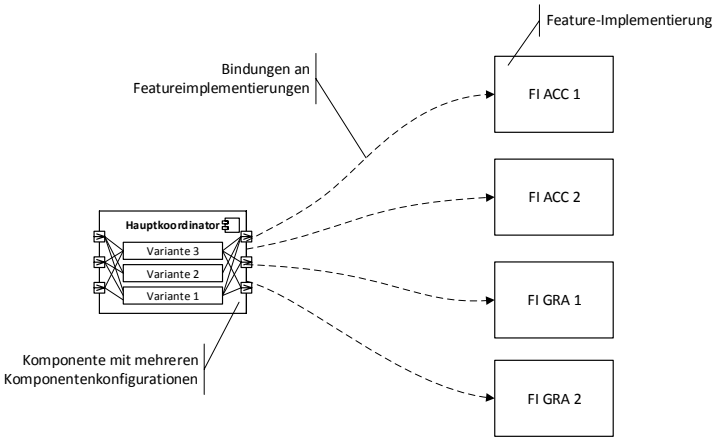


Abbildung 5.14: Reportsicht aus dem Blickwinkel einer Komponente

Die Assoziation ist darum als Komposition näher spezifiziert, deren Kardinalität auf Seite der Feature-Implementierung eine 1 ist. Da für die Realisierung eines Features eine Menge von Komponenten an verschiedene Parts gebunden werden muss, ist die Kardinalität auf Seite der Implementierungsbindung entsprechend angegeben.

Die Implementierungsbindung repräsentiert eine Assoziationsklasse, die die Beziehung einer Komponentenkonfiguration mit genau einem Part im Modell spezifiziert. Das Besondere an der Assoziation zwischen Implementierungsbindung und Komponentenkonfiguration (`konfBindung`) ist, dass sie nicht explizit modelliert sondern abgeleitet wird. Die Ableitung erfolgt über eine zusätzliche abstrakte Klasse (`Bindungselement`). Sowohl die Komponente wie auch die Komponentenkonfiguration sind Spezialisierungen dieser Klasse.

Diese umständlich anmutende Konstruktion ist für die eigentliche Bindung nicht notwendig. Die Modellierung der `konfBindung` würde zur Beschreibung der Bindung ausreichen. Der Vorteil dieser Konstruktion wird bei der Modellierung der Feature-Implementierungen sichtbar. Wenn der Architekt explizit eine Komponentenkonfiguration für jede Implementierungsbindung angeben müsste, würde er bei Komponenten mit vielen möglichen Konfigurationen ebenso viele Feature-Implementierungen modellieren. Durch

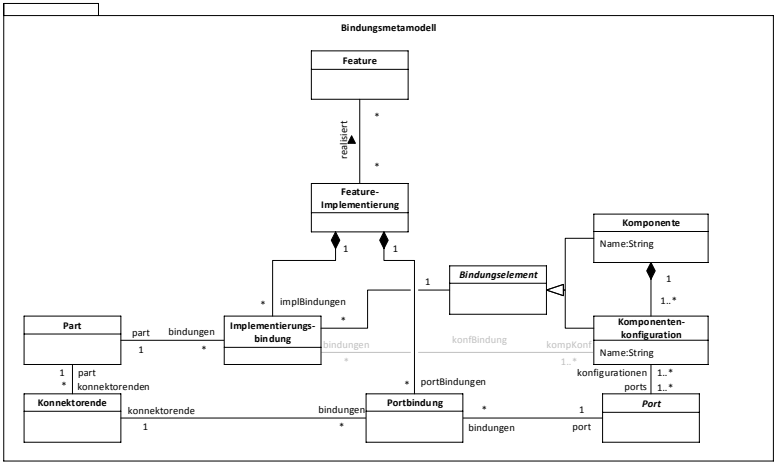


Abbildung 5.15: Abstrakte Syntax des Bindungs-Metamodells

die Einführung der Bindungselement-Klasse kann der Architekt wählen, ob er die Komponentenkonfiguration oder die Komponente auswählt.

Komponentenkonfigurationen können in verschiedenen Feature-Implementierungen eingesetzt werden. Das Gleiche gilt für Parts. Die Assoziationen zwischen Komponentenkonfiguration und Implementierungsbindung bzw. Part und Implementierungsbindung beschreiben darum eine 1-zu-*-Beziehung.

Der zweite Bestandteil einer Feature-Implementierung ist die Beschreibung, welche Ports der Komponenten an die Konnektoren des Strukturfragments gebunden werden. Die Klasse `Portbindung` beschreibt diese Modelleigenschaft. Eine konkrete Portbindung ist wie auch die Implementierungsbindung nur im Kontext einer Feature-Implementierung existent. In jeder Feature-Implementierung werden mehrere Ports an jeweils ein Konnektorende gebunden. Die entsprechende Assoziation spezifiziert eine 1-zu-*-Beziehung als Komposition.

Zwischen Port und Konnektorende muss im Kontext einer Feature-Implementierung eine 1-zu-1 Beziehung existieren. Im Kontext der gesamten Produktlinie können Ports an verschiedene Konnektoren gebunden werden bzw. Konnektorenden mit verschiedenen Ports in Beziehung stehen. Die Assoziationen im Metamodell zwischen Port und Portbindung bzw.

Konnektorende und Portbindung realisieren darum jeweils eine 1-zu-*-Beziehung.

Randbedingungen in OCL

Eine wesentliche Einschränkung einer Feature-Implementierung ist, dass die Parts und Konnektorenenden, die gebunden werden, zum selben Strukturfragment gehören. Diese Randbedingung wird durch den OCL-Ausdruck 5.10 spezifiziert.

```
context Feature-Implementierung
inv: self.implBindungen->forAll(ib1, ib2:
    Implementierungbindung |
          ib1 <> ib2 implies
            ib1.part.strukturfragment = ib2.part.
                strukturfragment)
inv: self.portBindungen->forAll(pb1, pb2:Portbindung |
          pb1 <> pb2 implies
            pb1.konnektorende.konnektor.strukturfragment =
            pb2.konnektorende.konnektor.strukturfragment)
inv: self.implBindungen->any().part.strukturfragment =
     self.portBindungen->any().konnektorende.konnektor.
         strukturfragment
```

OCL-Ausdruck 5.10: Ein Strukturfragment pro Feature-Implementierung

Bei den Komponentenkonfigurationen und Ports muss sichergestellt sein, dass sie an entsprechende zusammengehörige Elemente eines Strukturfragments gebunden werden. Das bedeutet, dass ein Port nur an das Konnektorende gebunden werden kann, das mit dem Part in Beziehung steht, an den die Komponentenkonfiguration gebunden ist. Diese Randbedingung wird im OCL-Ausdruck 5.11 ausgedrückt.

```
context Feature-Implementierung
inv: self.implBindungen->forAll(ib : Implementierungsbindung
     |
          self.portBindungen->select(pb : Portbindung |
               pb.port in ib.kompKonf.ports)
          .konnektorende in ip.part.konnektorenden)
```

OCL-Ausdruck 5.11: Einschränkung der Portbindung durch Implementierungsbindung

Darüber hinaus gilt, dass alle in einer Feature-Implementierung gebundenen Ports in den gebundenen Komponentenkonfigurationen enthalten sind.

```
context Feature-Implementierung
inv: self.implBindungen.komKonf.ports.flatten()
                                .includesAll(self.
                                    portbindungen.port)
```

OCL-Ausdruck 5.12: Einschränkung der Portnutzung bezüglich Komponentenkonfiguration

Die Ableitung der Assoziation `konfBindung` (s. OCL-Ausdruck 5.13) kann entweder direkt durch die Assoziation zum `Bindungselement` beschrieben werden, wenn das konkrete Bindungselement eine `Komponentenkonfiguration` ist. Ist das Bindungselement allerdings eine `Komponente`, wird zusätzlich die Kompositionsassoziation zwischen `Komponente` und `Komponentenkonfiguration` benötigt. Der Architekt wählt aus, ob er an den Part eine Komponente oder Komponentenkonfiguration binden will. Die abgeleitete Assoziation schränkt ein, dass eine Komponentenkonfiguration das tatsächlich gebundene Element ist.

```
context Implementierungsbindung::konfBindung : Set(
    Komponentenkonfiguration)
derive: if self.bindungselement.oclIsTypeOf(
    Komponentenkonfiguration)
        then self.bindungselement
        else self.bindungselement.komponentenkonfiguration
        endif
```

OCL-Ausdruck 5.13: Ableitung der Assoziation `konfBindung`

5.2 Konfigurationsvarianten und Produktarchitekturen

Die Hauptaufgabe des Software-Produklinienmodells besteht darin, die verschiedenen Produkte in einem Modell zu beschreiben. Die in den bisherigen Abschnitten gezeigten Metamodell-Teile zeigen, wie die einzelnen Modellteile aufgebaut sind. Abbildung 5.1 zeigt darüber hinaus auch, dass das Metamodell ebenso beschreibt, wie konkrete Produktarchitekturen aufgebaut sind. In dem nun folgenden Teil wird gezeigt, welche Eigenschaften das Softwareproduktlinienmodell zusätzlich haben muss, damit Produktarchitekturen abgeleitet werden können.

Der letzte Abschnitt im Kapitel 4 beschrieb, dass nach der Vorgabe einer Feature-Konfiguration geeignete Mengen von Feature-Implementierungen zusammengestellt werden. Diese werden zu Produktarchitekturen verschmolzen. Durch unterschiedliche Lösungsmengen und jeweils verschiedene Verschmelzungsmöglichkeiten sind mehrere Varianten von Produktarchitekturen pro Feature-Konfiguration ableitbar. Die verschiedenen Architekturvarianten müssen im Modell berücksichtigt werden. Das Metamodell wird um diese Möglichkeit erweitert.

Außerdem werden die Beschreibungen der im vorherigen Abschnitt noch unterspezifizierten Assoziationen, die in den Abbildungen 5.7 und 5.10 farblich hervorgehobenen sind, vervollständigt. Die definierten OCL-Ausdrücke, die diese Assoziationen referenzieren, behalten ihre Gültigkeit. Am Ende des Abschnitts wird ein kompletter Überblick über das gesamte Metamodell präsentiert.

5.2.1 Konkrete Syntax einer Produktarchitektur

Die Darstellung einer Produktarchitektur ist in Abbildung 5.16 exemplarisch dargestellt. Die konkrete Syntax entspricht dabei weitgehend einem Komponentendiagramm aus der UML. Komponenten werden als Rechtecke dargestellt, die mit dem aus der UML bekannten Komponenten-Symbol markiert sind. Ports werden als kleine Quadrate auf den Rändern der Komponenten dargestellt. Die Pfeile symbolisieren wie in Aschnitt 5.1.2 beschrieben die Art des Ports (Eingangs- oder Ausgangsport).

Auf die eingeführten Darstellungen zum Beispiel von Komponenten mit Konfigurationsvarianten wird verzichtet, da diese der Repräsentation der Variabilität dienten. In einer Produktarchitektur ist jede Form der Variabilität aufgelöst. Das bedeutet auch, dass nicht genutzte Ports ausgeblendet werden.

Das Diagramm wird durch Angabe der Feature-Konfiguration erweitert, also der Menge der ausgewählten Features, die durch diese Produktarchitektur realisiert werden. Für die Darstellung wird das in Abschnitt 4.2.1 vorgestellte Konzept genutzt. Da auch im Feature-Diagramm die Variabilität aufgelöst wurde, sind auch die Gruppierungs- und Querbeziehungs-Informationen nicht mehr darzustellen. Die monohierarchische Anordnung der ausgewählten Features bleibt erhalten. Darum eignet sich eine Baumstruktur als Darstellung einer Feature-Konfiguration.

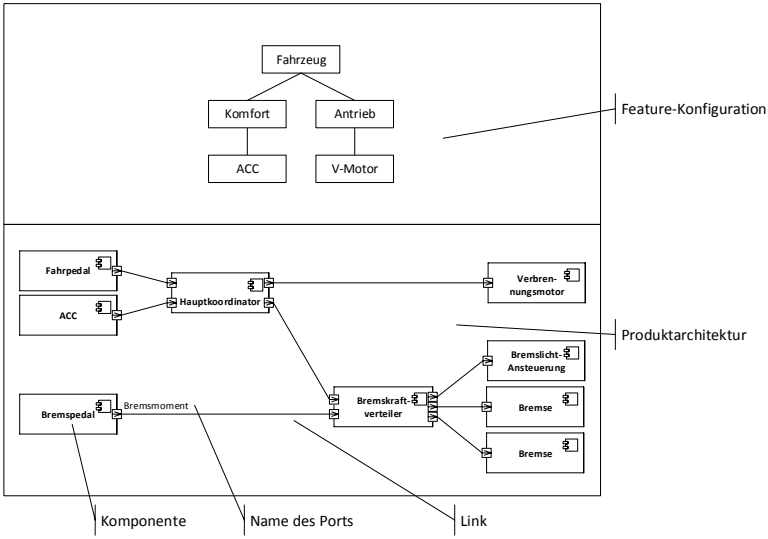


Abbildung 5.16: Darstellung einer Feature-Konfiguration und der entsprechenden Produktarchitektur (die meisten Portnamen sind ausgeblendet)

5.2.2 Abstrakte Syntax einer Produktarchitektur

Die Beschreibung der abstrakten Syntax, die in Abbildung 5.17 dargestellt ist, beginnt bei der Klasse `Feature-Konfiguration`. Eine Feature-Konfiguration repräsentiert die Menge der im *application engineering*-Prozess ausgewählten Features. Da natürlich die Features in mehreren Feature-Konfigurationen ausgewählt werden können, ist die Beziehung zwischen den Klassen mit einer *-zu-*-Kardinalität ausgestattet.

Für die eingangs erwähnte Berücksichtigung verschiedener Produktarchitektur-Varianten im Modell, die dieselbe Feature-Konfiguration realisieren, wird im Metamodell die Klasse `Konfigurationsvariante` eingeführt. Jede Feature-Konfiguration kann durch mehrere Konfigurationsvarianten realisiert werden. Umgekehrt realisiert jede Konfigurationsvariante genau eine Feature-Konfiguration.

Alle Konfigurationsvarianten sind Kandidaten für eine Produktarchitektur. Der Architekt muss aus den Konfigurationsvarianten, die die gewählte

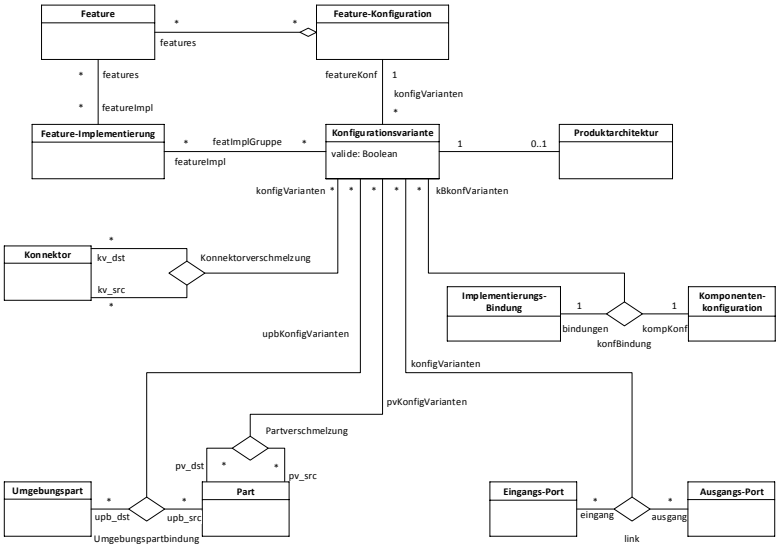


Abbildung 5.17: Abstrakte Syntax einer Produktarchitektur

Feature-Konfiguration realisieren, unter Berücksichtigung der technischen Randbedingungen eine geeignete auswählen (vgl. Abschnitt 4.4). Nicht jede Konfigurationsvariante kann als Produktarchitektur eingesetzt werden. Um diese markieren zu können, enthält die Klasse das Attribut `valide`. Nur als valide markierte Konfigurationsvarianten lassen sich als Produktarchitektur einsetzen.

Durch die Angabe einer Feature-Konfiguration lassen sich Mengen von Feature-Implementierungen ableiten, die die entsprechenden Features realisieren (s. Abbildung 5.15). Für jede Menge wird eine Konfigurationsvariante abgeleitet. Für die Verschmelzung der Feature-Implementierungen einer Menge werden die Regeln aus Abschnitt 5.1.3 angewendet. Dabei werden die Assoziationen `Umgebungspartbindung`, `Partverschmelzung` und `Konnektorverschmelzung` berücksichtigt. Diese waren in Abbildung 5.10 hellgrau markiert, weil die konkrete Syntax noch unterspezifiziert war. Im Vergleich mit Abbildung 5.17 wird klar, welche Information noch notwendig ist: die Beziehung zur Konfigurationsvariante. Darum sind die drei genannten Assoziationen als ternäre Assoziationen spezifiziert. Hierüber

lässt sich die konkrete Verschmelzung im Kontext einer Konfigurationsvariante beschreiben. Die Assoziation `link` ist aus vergleichbarem Grund als eine ternäre Assoziation spezifiziert (s. hierzu Abbildung 5.7). Die letzte noch zu erwähnende ternäre Assoziation ist die `konfBindung`. Die in Abbildung 5.15 noch binär dargestellte Assoziation stellt sicher, dass nur eine Komponentenkonfiguration an einen Part gebunden wird. Insbesondere dann, wenn der Architekt bei der Implementierungsbindung eine Komponente angibt. Einen Überblick über das gesamte Metamodell bietet die Abbildung 5.18.

5.2.3 Randbedingungen in OCL

Die OCL-Ausdrücke, die im Folgenden definiert werden, beschreiben Randbedingungen, die das Modell erfüllt, wenn die im Abschnitt 4.4 beschriebenen Phasen durchlaufen sind. Das bedeutet, dass die Eigenschaften der betrachteten Modellteile ermöglichen, dass nicht nur die Variabilität der Softwareproduktlinie sondern insbesondere auch konkrete Produkte beschrieben werden.

Das zentrale Element der folgenden Constraints ist die Konfigurationsvariante. Die beschriebenen Invarianten betreffen die in Abbildung 5.17 gezeigten ternären Assoziationen. Um die Navigation durch das Modell in den OCL-Ausdrücken besser nachvollziehen zu können, wird in Abbildung 5.18 ein Überblick über das gesamte Metamodell dargestellt.

Die relativ komplexen Ausdrücke sind eine Folge der Verwendung mehrerer ternärer Assoziationen im Metamodell. Constraints bezüglich einer ternären Assoziation sind nicht so einfach auszudrücken, wie das mit binären Assoziationen möglich ist. Erschwerend kommt hinzu, dass teilweise mehrere ternäre Assoziationen an einem Constraint beteiligt sind.

Durch eine Änderung der abstrakten Syntax wäre eine Vereinfachung der OCL Constraints möglich gewesen [CW02]. Auf diese Änderung wurde jedoch bewusst verzichtet, da zusätzliche Metamodell-Elemente die Komplexität in die abstrakte Syntax verschoben hätten.

Der erste OCL-Ausdruck (5.14) steht im Kontext einer Konfigurationsvariante. Er schreibt vor, dass die Featureimplementierungen, die mit der Konfigurationsvariante assoziiert sind, gerade die Features realisieren, die in der mit derselben Konfigurationsvariante assoziierten Featurekonfiguration enthalten sind.

```
context Konfigurationsvariante
inv: self.fetureImpl.features = self.featureKonf.features
```

OCL-Ausdruck 5.14: Einschränkung der Featureimplementierungen im Kontext einer Konfigurationsvariante

Das zweite Constraint in diesem Abschnitt beschreibt die eindeutige Zuordnung einer Komponentenkonfiguration zu einem Part im Kontext einer Konfigurationsvariante. Da im Metamodell bereits eindeutig ein Part mit einer Implementierungsbindung assoziiert ist, reicht es aus, im OCL-Ausdruck die Eigenschaften der Assoziation `konfBindung` einzuschränken. Wenn im Bindungsschritt des *domain engineering*s eine Komponentenkonfiguration an einen Part gebunden wird, muss genau diese Komponentenkonfiguration in der ternären Assoziation verbunden sein. Ist eine Komponente an einen Part gebunden, muss genau eine Komponentenkonfiguration der gebundenen Komponente im Kontext der Konfigurationsvariante genutzt werden.
In beiden Fällen ist notwendig, dass die Ports, die über eine Portbindung an ein entsprechendes Konnektorende gebunden sind, mit der gebundenen Komponentenkonfiguration assoziiert sind. Diese Portbindung ist Teil der jeweils betrachteten Featureimplementierung.
Der folgende OCL-Ausdruck spezifiziert diese Anforderung an das Modell über eine Invariante bezüglich der Assoziation `konfBindung`.
Deutlich komplexer gestaltet sich die Beschreibung der Bedingungen, die das Modell an einem `link` erfüllen muss. In den OCL-Ausdrücken 5.16 und 5.17 sind zwei Invarianten definiert. Bei der ersten Invariante werden die Links betrachtet, die im Kontext einer Konfigurationsvariante an einem Eingangsport enden. In dem Fall muss die Anzahl der eingehenden Links zwischen den erlaubten Grenzen liegen. Die Grenzen werden durch die Attribute `minLinks` und `maxLinks` des Eingangsports spezifiziert. Außerdem muss der Port am anderen Ende des Links ein `Ausgangsport` sein, der den gleichen Port-Typen hat wie der Eingangsport. Der letzte wichtige Teil des Constraints setzt die Links zwischen Komponenten mit Konnektoren zwischen Parts in Beziehung. Ein Link darf nur dann zwischen zwei Komponenten existieren, wenn zwischen den Parts des Strukturfragments, an die die Komponenten gebunden sind, ein Konnektor existiert.

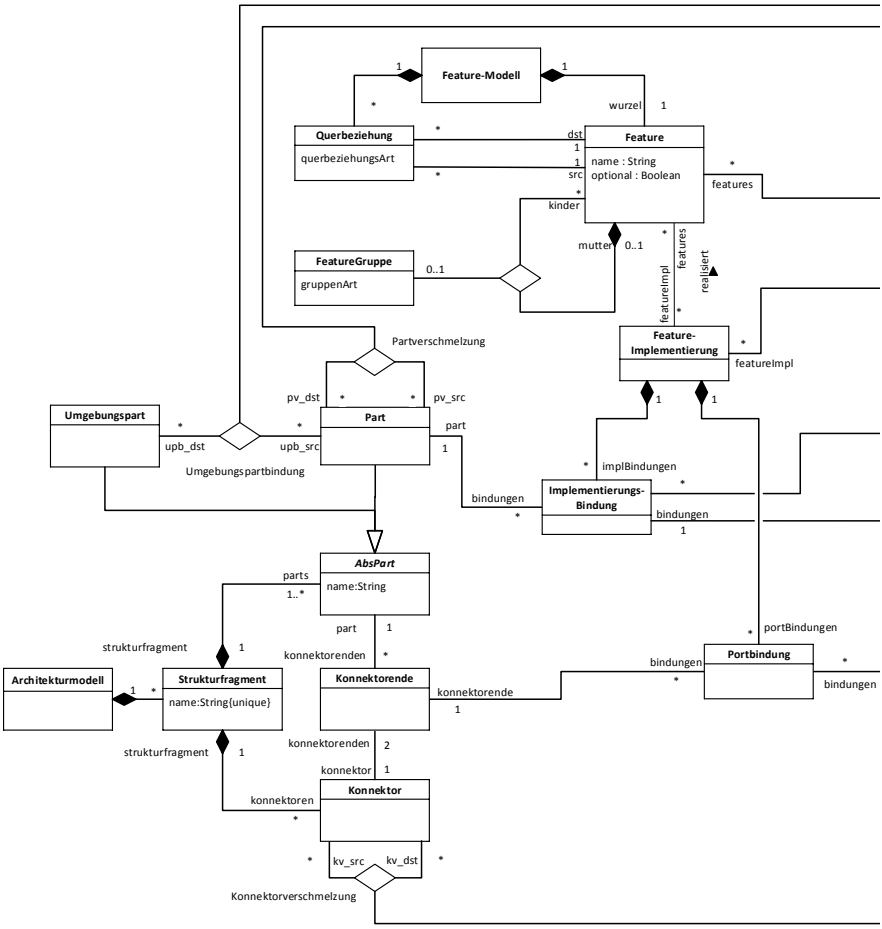
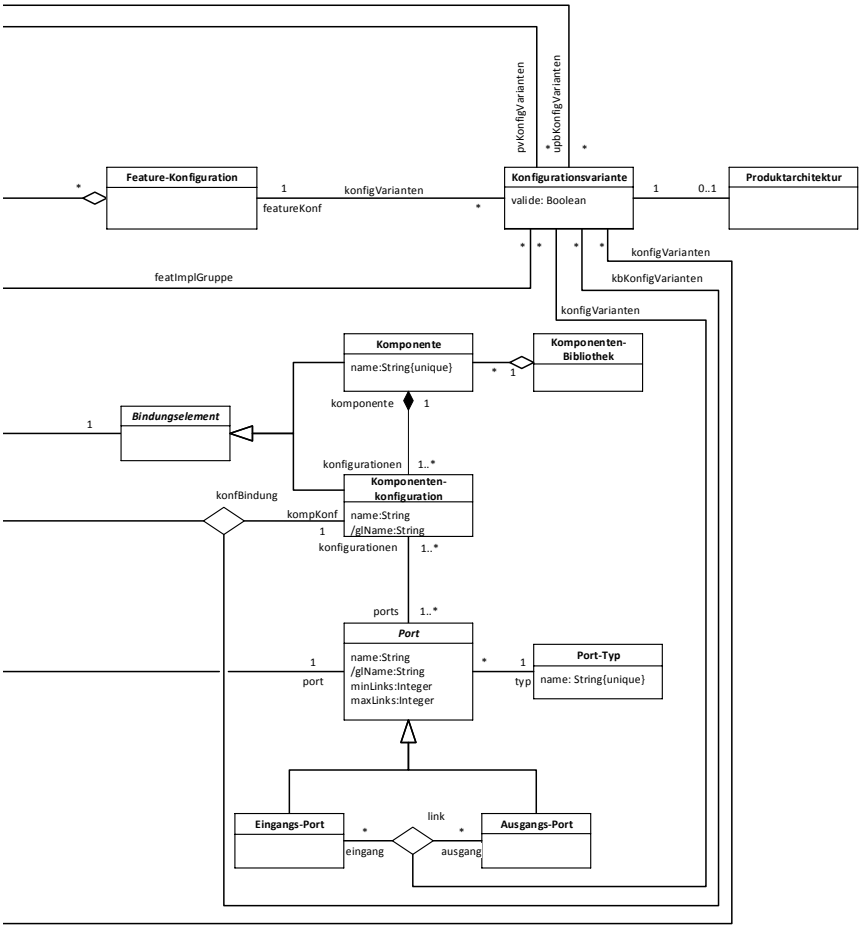


Abbildung 5.18: Das gesamte Metamodell zur Beschreibung eines Produktlinienmodells


pvKonfigVarianten
upbKonfigVarianten
Feature-Konfiguration
1
konfigVarianten
featureKonf
Konfigurationsvariante
valide: Boolean
1
0..1
Produktarchitektur
konfigVarianten
featImplGruppe
kbKonfigVarianten
konfigVarianten
Komponente
name:String{unique}
Komponenten-Bibliothek
komponente
1
Bindungselement
konfigurationen
1..*
Komponenten-konfiguration
name:String
/glName:String
konfBindung
kompKonf
1
konfigurationen
1..*
ports
1..*
Port
name:String
/glName:String
minLinks:Integer
maxLinks:Integer
1
port
Port-Typ
name: String{unique}
typ
link
Eingangs-Port
eingang
ausgang
Ausgangs-Port

```
context Konfigurationsvariante
inv: self.featureImpl.implBindung->forAll( ib |
       if ib.bindungselement.oclIsTypeOf(
           Komponentenkonfiguration)
       then
         ib.bindungselement->select(kk | konfBindung(self,ib
             ,kk)
         and
         ib.part.konnektorenden->exists(ke | ke.bindungen->
             exists(pb | pb.port in kk.port)))
       else
         ib.bindungselement.konfigurationen->any(kk |
             konfBindung(self,ib,kk)
         and
         ib.part.konnektorenden->exists(ke | ke.bindungen->
             exists(pb | pb.port in kk.port)))
       endif)
```

OCL-Ausdruck 5.15: Spezifikation einer eineindeutigen Zuordnung einer Komponentenkonfiguration zu einer Implementierungsbindung

```
context Konfigurationsvariante
inv: self.featureImpl.portbindungen->select( pb1 |
          pb1.port.oclIsTypeOf(Eingangsport)).port->forAll(
              ep |
              self.featureImpl.portbindungen->select( pb2 |
                   pb2.port.oclIsTypeOf(Ausgangsport)).port
                       ->select(ap |
                      link(self,ep,ap)
                      and
                      ep.typ = ap.typ
                      and
                      self.featureimpl.portbindungen->exists
                          (
                          pb3, pb4 | pb3.konnektorende.
                              konnektor.konnektorende->any(
                              ke| pb4 in
                              ke.bindungen)
```

```
                        and
                        pb3 <> pb4
                        and
                        pb3.port = ep and pb4.port = ap)
                    )->size() >= ep.minLinks
            and
            self.featureImpl.portbindungen->select( pb2 |
                 pb2.port.oclIsTypeOf(Ausgangsport)).port
                     ->select(ap |
                    link(self,ep,ap)
                    and
                    ep.typ = ap.typ
                    and
                    self.featureimpl.portbindungen->exists
                        (
                      pb3, pb4 | pb3.konnektorende.
                          konnektor.konnektorende->any(ke|
                           pb4 in
                          ke.bindungen)
                      and
                      pb3 <> pb4
                      and
                      pb3.port = ep and pb4.port = ap)
                      )->size() <= ep.maxLinks)
```

OCL-Ausdruck 5.16: Spezifikation zusätzlicher Randbedingungen hinsichtlich der Links im Modell

Die zweite Invariante beschreibt vergleichbare Randbedingungen aus der Sicht eines Ausgangsports im Kontext einer Konfigurationsvariante. Dementsprechend ist diese Invariante mit vertauschten Rollen bezüglich der Ports genauso aufgebaut wie die erste.

```
context Konfigurationsvariante
inv: self.featureImpl.portbindungen->select( pb1 |
          pb1.port.oclIsTypeOf(Ausgangsport)).port->forAll(
              ep |
              self.featureImpl.portbindungen->select( pb2 |
                   pb2.port.oclIsTypeOf(Eingangsport)).port
                       ->select(ap |
                      link(self,ep,ap)
                      and
```

```
                    ep.typ = ap.typ
                    and
                    self.featureimpl.portbindungen->exists
                        (
                        pb3, pb4 | pb3.konnektorende.
                            konnektor.konnektorende->any(
                            ke| pb4 in
                            ke.bindungen)
                        and
                        pb3 <> pb4
                        and
                        pb3.port = ep and pb4.port = ap)
                    )->size() >= ap.minLinks
            and
            self.featureImpl.portbindungen->select( pb2 |
                 pb2.port.oclIsTypeOf(Eingangsport)).port
                     ->select(ap |
                    link(self,ep,ap)
                    and
                    ep.typ = ap.typ
                    and
                    self.featureimpl.portbindungen->exists
                        (
                        pb3, pb4 | pb3.konnektorende.
                            konnektor.konnektorende->any(
                            ke| pb4 in
                            ke.bindungen)
                        and
                        pb3 <> pb4
                        and
                        pb3.port = ep and pb4.port = ap)
                    )->size() <= ap.maxLinks)
```

OCL-Ausdruck 5.17: Spezifikation zusätzlicher Randbedingungen hinsichtlich der Links im Modell

Um Parts verschmelzen zu können, müssen die in OCL-Ausdruck 5.18 definierten Bedingungen erfüllt werden. Die zu verschmelzenden Parts gehören zu verschiedenen Strukturfragmenten. Die an die Parts gebundenen Komponentenkonfigurationen müssen beide im Kontext der Konfigurationsvariante an einer `konfBindung` beteiligt sein. Wenn die gebundenen

Komponentenkonfigurationen identisch sind, können die entsprechenden Parts an einer Partverschmelzung beteiligt sein.

```
context Konfigurationsvariante
inv: self.featureImpl.implBindungen->forAll(ib1 |
          self.featureImpl.implBindungen->select( ib2 |
          Partverschmelzung(self, ib1.part, ib2.part))->
              forAll(
               konfBindung(self, ib1.bindungselement, ib1)
               and
               konfBindung(self, ib2.bindungselement, ib2)
               and
               ib1.bindungselement = ib2.bindungselement
               and
               ib1.part.strukturfragment <> ib2.part.
                   strukturfragment))
```

OCL-Ausdruck 5.18: Invariante bezüglich der ternären Assoziation `Partverschmelzung`

Die ternäre Assoziation `Umgebungspartbindung` darf nur existieren, wenn die in OCL-Ausdruck 5.19 spezifizierten Randbedingungen gelten.
Ein Umgebungspart gibt an, wie die strukturelle Umgebung eines verschmolzenen Parts aussehen muss. Im Umkehrschluss bedeutet das, dass die Nachbar-Parts eines Parts und eines Umgebungsparts, die über eine Umgebungspartbindung in Beziehung stehen, miteinander verschmolzen sind. Der Kontext dieser Partverschmelzung muss derselbe sein wie in der entsprechenden Umgebungspartbindung. Wie bei der `Partverschmelzung` müssen die betrachteten Elemente `Part` und `Umgebungspart` zu verschiedenen Strukturfragmenten gehören.

```
context Konfigurationsvariante
inv: self.featureImpl.implBindungen->forAll( ib |
         Strukturfragment.allInstances(sf|
           sf.parts->select(up |
             up.oclIsTypeOf(Umgebungspart)
             and
             Umgebunbspartbindung(self, ib.part, up))->
                 forAll(
               ib.part.strukturfragment <> up.
                   strukturfragment
               and
```

```
                up.konnektorenden->exists(ke1 |
                  ke1.konnektor.konnektorende->exists(ke2 |
                    ke2 <> ke1
                    and
                    ib.part.konnektorenden->exists(ke3 |
                      ke3.konnektor.konnektorende->exists(ke4
                          |
                        ke3 <> ke4
                        and
                        Partverschmelzung(self,ke2.part,ke4.
                            part)
        ) ) ) ) ) ) )
```

OCL-Ausdruck 5.19: Invariante bezüglich der ternären Assoziation `Umgebungspartbindung`

Wenn aus zwei verschiedenen Strukturfragmenten Konnektoren im Kontext einer Konfigurationsvariante miteinander verschmolzen werden, muss die in OCL-Ausdruck 5.20 spezifizierte Randbedingung erfüllt sein. Der Ausdruck besagt, dass die Parts oder Umgebungsparts an den Enden eines Konnektors mit den Parts oder Umgebungsparts an den Enden des anderen Konnektors paarweise verschmolzen beziehungsweise gebunden sind. Die entsprechenden Partverschmelzungen und Umgebungspartbindungen stehen mit derselben Konfigurationsvariante in Beziehung wie die Konnektorverschmelzung selbst. Das impliziert, dass auch die verschmolzenen Konnektoren zu verschiedenen Strukturfragmenten gehören.

```
context Konfigurationsvariante
inv: self.featureImpl.portBindungen->forAll( pb1, pb2 |
          pb1 <> pb2
          and
          pb1.konnektorende.konnektor = pb2.konnektorende.
              konnektor
          and
          self.featureImpl.portBindungen->exists( pb3, pb4 |
            pb3 <> pb4
            and
            pb3.konnektorende.konnektor = pb4.konnektorende.
                konnektor
            and
            p1.port = p3.port
            and
```

```
            p2.port = p4.port
            and
            Konnektorverschmelzung(self, pb1.konnektorende.
                konnektor, pb3.konnektorende.konnektor)
            )
          )
```

OCL-Ausdruck 5.20: Invariante bezüglich der ternären Assoziation `Konnektorverschmelzung`

5.3 Algorithmische Spezifikation weiterer Modelleigenschaften

Die im letzten Abschnitt definierten OCL-Ausdrücke spezifizieren die meisten Randbedingungen an Softwareproduktlinienmodelle auf Basis des vorgestellten Konzepts. Für einige andere Bedingungen ist OCL nicht geeignet. Die Modelleigenschaften der Beziehungen zwischen den Objekten der Klassen Konfigurationsvariante, Featureimplementierung und Featurekonfiguration lassen sich relativ leicht durch einen Algorithmus definieren. Dieser Algorithmus dient nicht dazu, die Phasen aus Abschnitt 4.4 zu formalisieren. Stattdessen beschreibt er eine weitere Eigenschaft des Modells nach Durchlauf der Phasen – die Anzahl der Konfigurationsvarianten, die zu einer Feature-Konfiguration gehören.

Die Phasen *Implementierungsfilterung* und *Gruppierung* erzeugen nach Vorgabe einer Featurekonfiguration Mengen von Featureimplementierungen. Die Implementierungsfilterung bestimmt eine Menge von Featureimplementierungen, aus der kein Element ein Feature realisiert, das nicht in der Featurekonfiguration enthalten ist. In der Gruppierungsphase werden aus der Potenzmenge dieser gefilterten Featureimplementierungsmenge Teilmengen für die Nutzung in Konfigurationsvarianten gefunden. Jede dieser Teilmengen hat die Eigenschaft, dass jedes Feature der Featurekonfiguration durch genau eine Featureimplementierung dieser Teilmenge realisiert wird. Jede Konfigurationsvariante, die über die Assoziation *konfigVarianten* mit einer Featurekonfiguration in Beziehung steht, beschreibt ein *Matching* zwischen Features und den Featureimplementierungen. Über die Assoziation *featureImplGruppe* werden die entsprechenden Featureimplementierungen referenziert. Dabei ist wichtig, dass alle möglichen Kombinationen von Featureimplementierungen berücksichtigt werden.

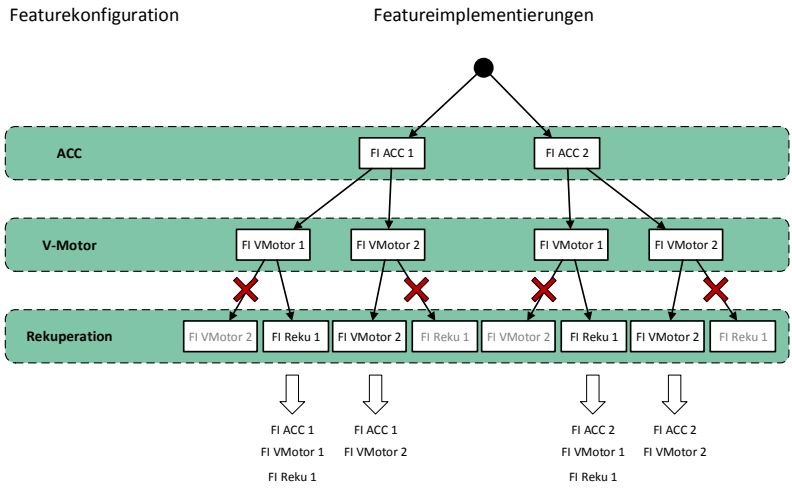


Abbildung 5.19: Zum Auffinden aller Featureimplementierungskombinationen wird ein Baum aufgebaut

Der folgende Algorithmus beschreibt, wie die Matchings für die Konfigurationsvarianten anhand des Modells gefunden werden können. Die Kernidee des Algorithmus basiert auf der Erzeugung eines Baums, in dem jeder Pfad durch eine mögliche Lösung von Featureimplementierungen repräsentiert wird.
Jeder Ebene im Baum ist ein Feature der Featurekonfiguration zugeordnet. Beim Aufbau des Baums wird an jeden Knoten der nächsthöheren Ebene die Menge aller Featureimplementierungen gehängt, die das Feature der entsprechenden Ebene realisieren. Wenn eine Featureimplementierung ein Feature realisiert, das bereits durch einen Vorfahren realisiert wird, würde jeder Pfad durch den entsprechenden Knoten ein gültiges Ergebnis liefern. Der Knoten wird in dem Fall nicht an den Baum gehängt. In Abbildung 5.19 wird das durch rote Kreuze an diesen Stellen angedeutet. Die Featureimplementierung *FI VMotor 2* realisiert das Feature `Rekuperation`. Da es aber auch das Feature *V-Motor* realisiert, darf die entsprechende Featureimplementierung nicht an den Baum gehängt werden.
Das Beispiel zeigt vier mögliche Lösungen. Es ist zu beachten, dass ein Pfad, der mehrfach dieselbe Featureimplementierung enthält, kein Problem darstellt. In dem Fall realisiert die entsprechende Featureimplementierung

mehrere Features. Die Lösungsmenge enthält diese Featureimplementierung dann nur einmal.

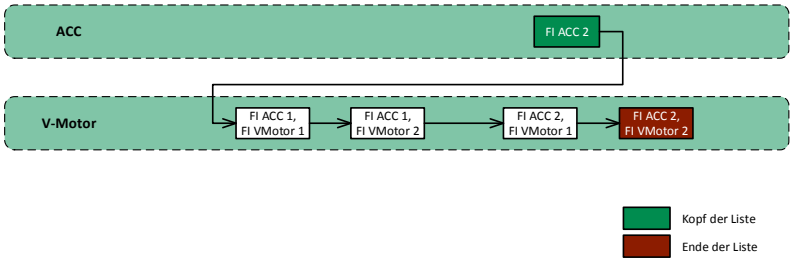


Abbildung 5.20: Schnappschuss der Liste während der Ausführung des Algorithmus

Um den Algorithmus einfacher zu gestalten, wird der Baum auf eine lineare Datenstruktur abgebildet. Abbildung 5.20 zeigt einen Schnappschuss der linearen Liste, die der Algorithmus konstruiert. Der Kopf der Liste repräsentiert den Knoten, dessen Kinder als nächstes erzeugt werden müssen. Die Kinder werden an das Ende der Liste angehängt. Die einzelnen Elemente der Liste speichern nicht nur die Featureimplementierung des entsprechenden Kindknotens im Baum, sondern auch die aller Ahnen. Dadurch muss die Beziehung zum Elternknoten nicht mehr gespeichert werden. Das bedeutet, dass der Kopf aus der Liste entfernt wird, sobald seine Kinder bestimmt wurden.

Ein Kind wird nur an die Liste angehängt, wenn die oben genannten Randbedingungen bezüglich der Menge der Featureimplementierungen nicht verletzt werden und zusätzlich die Matching Randbedingung erfüllt ist. Diese besagt, dass jedes Feature in einem Produkt nur durch genau eine Featureimplementierung realisiert werden darf. Um diese Bedingung prüfen zu können, werden alle Features der Featurekonfiguration, die noch nicht durch Featureimplementierungen realisiert sind, dem entsprechenden Element zugeordnet. Wenn eine neue Featureimplementierung bei der Kinderzeugung aufgenommen werden soll, müssen alle Features, die diese Featureimplementierung realisiert, in der Menge enthalten sein. Ist das nicht der Fall, wird die Randbedingung verletzt und das entsprechende Kindelement nicht in der Liste aufgenommen. Falls das Element in der Liste aufgenommen wird, müssen alle Features, die die neue Featureimple-

mentierung realisiert, aus der Liste der noch nicht realisierten Features entfernt werden.
Wenn alle Ebenen, die ein Feature der Featurekonfiguration repräsentieren, vollständig bearbeitet wurden, enthält die Liste nur noch Elemente, die gültige Mengen von Featureimplementierungen enthalten. Die gesuchte Anzahl an Konfigurationsvarianten mit verschiedenen Mengen von Featureimplementierungen (`featureImpl`) in Bezug auf eine Feature-Konfiguration ist gerade die Größe dieser Liste. Das Listing in Pseudocode 1 repräsentiert den vorgestellten Algorithmus.

5.4 Zusammenfassung

Dieses Kapitel spezifiziert die Modellierungssprache, mit der sich ein Software-Produktlinienmodell nach dem in Kapitel 4 vorgestellten Konzept erstellen lässt. Das Meta-Modell einer Softwareproduktlinie wird aus einzelnen Teilen zusammengesetzt. Der Aufbau eines Featuremodells, eines Architekturmodells mit seinen Strukturfragmenten und der Komponentenbibliothek werden jeweils durch einzelne Meta-Modellteile spezifiziert. Durch Beschreibung der Bindungen werden diese Teile zu einem Ganzen zusammengefügt.
Zusätzlich zur Spezifikation einer abstrakten Syntax in Form von UML-Klassendiagrammen wird zu jedem Teil noch eine konkrete Syntax vorgeschlagen. OCL-Constraints und Algorithmen geben weitere Randbedingungen an, die das Modell der Softwareproduktlinie einhalten muss. Ein Diagramm, das die komplette abstrakte Syntax enthält, ist auf Seite 116 des Kapitels in Abbildung 5.18 dargestellt.

Pseudocode 1 Algorithmus zur Bestimmung der Anzahl an Konfigurationsvarianten zu einer Featurekonfiguration

```
function CountMatchings(fc)                    ▷ fc ist die ausgewählte Featurekonfiguration
    head.toImplement ← fc.features
    head.featureImpl ← ∅
    head.noRealizedFeatures ← 0
    head.level ← 0
    list ← ()
    actLevel ← 0
    for all f ∈ fc.features do
        actLevel ← actLevel + 1
        while head.level < actLevel do
            for all fi ∈ f.featureImpl do
                node ← Copy(head)
                if AddFI(node, fi) = true then
                    list ← Append(list, node)
                end if
            end for
            RemoveHead(list)
            head ← GetHead(list)
        end while
    end for
    return list.size
end function

function AddFI(node, fi)
    if fi ∉ node.featureImpl then
        for all f ∈ fi.realizedFeatures do
            if f ∈ node.toImplement then
                node.toImplement ← node.toImplement \ {f}
                noRealizedFeatures ← noRealizedFeature + 1
            else
                return false
            end if
        end for
    end if
    node.level ← node.level + 1
    node.featureImpl ← node.featureImpl ∪ {fi}
    return true
end function
```

Kapitel 6

Beschreibung eines Modellierungstools

In dem nun folgenden Kapitel wird ein Werkzeugprototyp beschrieben, mit dem das letzte Ziel dieser Arbeit erreicht werden soll – die Demonstration der Anwendbarkeit des Konzepts (s. Abschnitt 3.3). Um den Umfang dieses Kapitels nicht unnötig wachsen zu lassen, wird auf eine umfassende Spezifikation oder gar komplette Dokumentation der Software verzichtet. Vielmehr soll hier nur eine Idee vermittelt werden, wie die wesentlichen Konzepte einer konkreten Werkzeugumsetzung aussehen können. An dieser Stelle sei erwähnt, dass zum Zeitpunkt des Verfassens dieser Dissertation der Prototyp noch nicht vollständig umgesetzt ist. Die noch fehlenden Teile werden in den Darstellungen durch Mockups ersetzt.
Der erste Teil des Kapitels beschreibt einige Anforderungen an das Tool. Im zweiten Teil folgt ein Überblick über die Lösungsstrategie, in der bereits fundamentale Designentscheidungen getroffen werden. Danach folgt eine detailliertere Beschreibung der Tool-Architektur hinsichtlich Struktur und Verhalten. Abschließend werden querschnittliche Konzepte beleuchtet.

6.1 Wesentliche Anforderungen

Die wesentliche Kernaufgabe der Software ist die Erstellung von Modellen, die konform sind zum im vorherigen Kapitel 5 beschriebenen Metamodell. Darüber hinaus muss der Entwicklungsprozess für die Entwicklung einer fragmentbasierten Softwareproduktlinie (s. Abschnitt 4.1) von der Software komplett unterstützt werden.
Das Werkzeug muss die abgeleiteten Architekturmodelle jeweils in instanziierbare Modelle transformieren können. Das in den vorherigen Kapiteln beschriebene Konzept beschreibt die Softwareartefakte nur in einer generalisierten Weise. Das Komponentenmodell beschreibt nur die für das

Konzept relevanten Eigenschaften. Das ermöglicht, die Architekturableitung unabhängig von konkreten Implementierungen durchzuführen.
Die abstrakten Komponentenbeschreibungen müssen mit konkreten Implementierungen in Beziehung gesetzt werden. Dieser Schritt entspricht im Grunde einer Modelltransformation bezüglich des MDA-Konzepts hin zu einer konkreteren Architektur (s. Kapitel 2).
Das entwickelte System ist ein Prototyp, anhand dessen die Umsetzbarkeit der Werkzeugunterstützung bei fragmentbasierter Architekturentwicklung in Softwareproduktlinien demonstriert werden soll. Darum werden viele Anforderungen, die üblicherweise an ein System für den produktiven Einsatz gestellt werden, bewusst ausgeblendet. Hierzu zählen Anforderungen, die zum Beispiel Performanz, Wartbarkeit, Sicherheit etc. betreffen.

6.2 Lösungsstrategie

In diesem Abschnitt werden einige wichtige Entscheidungen bzgl. des Architekturdesigns beschrieben. Die Software soll modular aufgebaut werden. Die Betrachtung der Anforderungen und des in Abschnitt 4.1 vorgestellte Entwicklungsprozesses legen eine Entscheidung bezüglich der Modulgrenzen nah.
Modularität soll nicht nur bei der Software erreicht werden, sondern auch bei den persistierten Daten. Eine einfache Lösung, die für den Prototypen vollkommen hinreichend ist, ist die Zerlegung des Softwareproduktlinienmodells in mehrere Dateien. Jedes Strukturfragment und jede Komponente wird in einer eigenen Datei beschrieben.
Um den Entwicklungsaufwand zu minimieren, wird bei der Entwicklung auf existierende Technik zurückgegriffen. Diese hat maßgeblich Einfluss auf die Architektur des Prototypen. Die Interaktion der verschiedenen Module wird über eine Middleware realisiert. Die Entscheidung hierfür fiel auf die Eclipse Plattform (RCP, s. [16d]).
Die konkrete Syntax der Modellelemente (s. Kapitel 5) erfordert grafische Editoren zur Erstellung eines Modells. Mit EMF/GMF basierten Tools stellt die Eclipse Foundation eine Technik zur Verfügung, mit der relativ komfortabel Editoren entwickelt werden können. Diese Editoren ermöglichen eine metamodellkonforme Modellierung der entsprechenden Modellelemente [16g]. Als Plug-ins können sie in eine RCP-Anwendung eingebunden werden.
Da der Schwerpunkt dieser Arbeit auf der Modellierung des *solution space* liegt, kann für die Modellierung des *problem space* das Tool *FeatureIDE* [Thü+14] eingesetzt werden. *FeatureIDE* ist ebenfalls als Eclipse

RCP entwickelt und lässt sich relativ einfach um weitere Funktionalität erweitern.

6.2.1 Umsetzung mit der Eclipse Rich Client Platform

Um verstehen zu können, wie die eingesetzten Technologien die Architektur beeinflussen, werden im Folgenden die Konzepte kurz erklärt. Als erstes wird das Rich Client Konzept der Eclipse-Plattform vorgestellt. Im Anschluss daran wird ein Blick auf die Erstellung und den Aufbau eines EMF/GMF-basierten Editors geworfen. Zum Schluss folgt die Beschreibung der Plug-ins aus *FeatureIDE*.

Eclipse RCP

Die Eclipse-Plattform bietet eine auf Equinox [16f] basierende Laufzeitumgebung. Die Architektur der Plattform ist ganz darauf ausgelegt, die Funktionalität des zu entwickelnden Systems über die Einbindung von Plug-ins zu realisieren. Ein Beispiel für eine entsprechende Systemarchitektur ist in Abbildung 6.1 dargestellt.

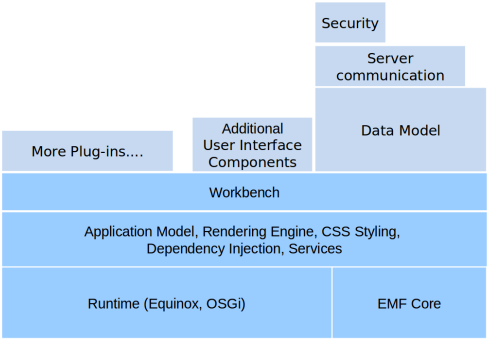


Abbildung 6.1: Typischer Aufbau eines Systems auf Basis der Eclipse-Plattform (Bildquelle: [Vog16])

Ein Plug-in kann Abhängigkeiten zu anderen Plug-ins haben aber auch so genannte *Extension Points* zur Verfügung stellen, über die die eigene Funktionalität erweitert werden kann. Die Bindung der Plug-ins erfolgt zu einem späten Zeitpunkt und wird durch die Plattform realisiert. Dafür kommt das Konzept der *dependency injection* zum Einsatz. Damit ist es möglich, die einzelnen Komponenten eines Systems relativ lose miteinander zu koppeln.

Ein weiterer Vorteil für die Entwicklung des Prototypen ist das *application model*. Das application model ist eine abstrakte Strukturbeschreibung der Applikation. Auf Basis dieser können mit ebenfalls integrierten User-Interface-Elementen komplexe grafische Oberflächen beschrieben werden. Außerdem existieren komfortable zu nutzende Möglichkeiten für die Dateiverwaltung.
Die Eclipse-Plattform eignet sich daher gut als Basis für den zu entwickelnden Prototypen. Weitere Informationen zu Eclipse RCP findet man in [16d; 16e; Vog16] oder im Buch „The Architecture of Eclipse“ von Lars Vogel [Vog15].

EMF/GMF

Das *Eclipse Modeling Framework* (EMF) ist ein Framework, mit dem es möglich ist, Tools zu konstruieren, die mit strukturierten Datenmodellen arbeiten [16c]. Aus einer Spezifikation der Datenstruktur werden Java Klassen generiert, die sowohl die Datenelemente selbst als auch Methoden zur Manipulation ebendieser bieten.
Das Graphical Modeling Framework (GMF) basiert unter anderem auf EMF und erweitert das Konzept um Techniken zum Generieren grafischer Editoren. Die Grafik in Abbildung 6.2 zeigt den Erzeugungsprozess solcher Editoren mittels modellbasiertem Ansatz.

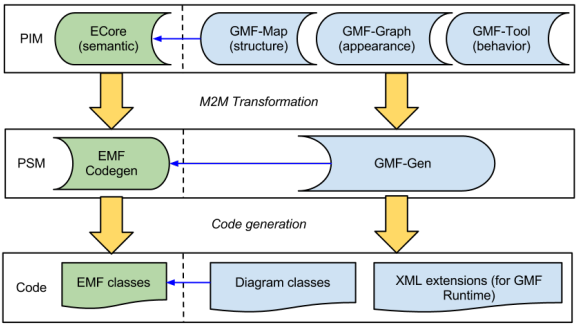


Abbildung 6.2: Modellbasierte Erzeugung eines grafischen Editors mit GMF (Bildquelle: [14a])

Die abstrakte Syntax des Datenmodells wird zuerst durch ein so genanntes ECore-Modell spezifiziert. Das Aussehen der verschiedenen Datenelemente im Diagramm wird durch das GMF-Graph-Modell beschrieben. Durch

die GMF-Map werden die Beziehungen zwischen den grafischen und den ECore-Elementen hergestellt. Für das Erstellen eines Diagramms mit dem Editor wird eine Toolbar integriert. Mit den Buttons in der Toolbar können neue Diagrammelemente erzeugt werden. Das dafür notwendige Verhalten wird im GMF-Tool-Modell spezifiziert. Auf Basis dieser vier Modelle kann ein grafischer Editor generiert werden.
In Kapitel 5 wird sowohl die abstrakte als auch die konkrete Syntax zur Modellierung eines SPL-Modells definiert. Der Großteil der Informationen zur Generierung eines entsprechenden GMF basierten Editors ist also vorhanden, weshalb GMF das ideale Werkzeug für die Entwicklung der Editor-Plug-ins darstellt. Weiterführende Informationen bzgl. EMF und GMF findet man hier: [Ste+08; 14b; 14a].
Die für die fachlichen Aspekte relevanten Komponenten sind bei GMF-basierten Editoren nach einem Model-View-Control-Muster (MVC) zusammengesetzt [Gam+04]. In der Regel werden drei Views erzeugt: Eine *Tool-Palette* enthält Steuerelemente für die Erzeugung einzelner Modell-Elemente, das eigentliche Diagramm wird in einer *Diagramm-Sicht* editiert, und zusätzliche Eigenschaften können in einer *Property-View* eingestellt werden. Das Besondere an der genutzten MVC-Variante ist, dass jeder Datenaustausch zwischen Modell und den Views durch den Controller realisiert wird (s. Abbildung 6.3).

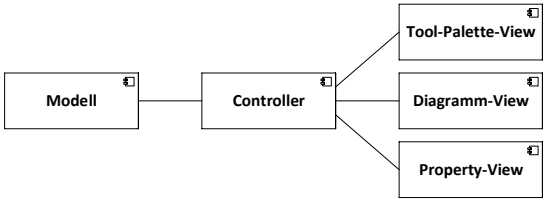


Abbildung 6.3: Struktur des Strukturfragmenteditors nach dem Model-View-Controller-Muster

FeatureIDE

FeatureIDE ist ein hauptsächlich an der Otto-von-Guericke-Universität Magdeburg entwickelte Entwicklungsumgebung für Feature-Orientierte Softwareentwicklung [Thü+14]. FeatureIDE umfasst mehrere Implementierungstechniken für Softwareproduktlinien. Die entscheidenden Vorteile

sind jedoch ein integrierter Featuremodell-Editor und ein Editor zur Erzeugung von Feature-Konfigurationen. Die Modellierungssprache gleicht der in Kapitel 5 definierten weitgehend, sodass der Editor für den Einsatz im Prototypen geeignet ist.
FeatureIDE ist ein Eclipse-Plug-in, das mit Extension Points für eigene Erweiterungen ausgestattet wurde. Abbildung 6.4 gibt einen Überblick über das Konzept zur Entwicklung konkreter Produkte. Mit der Entwicklung des Feature-Modells und der Ableitung verschiedener Feature-Konfigurationen unterstützt FeatureIDE bereits alle wesentlichen Schritte, die der Entwicklungsprozess in Abschnitt 4.1 für den *problem space* vorsieht.

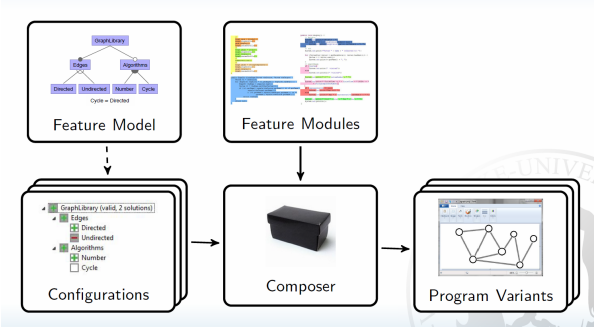


Abbildung 6.4: Überblick über das FeatureIDE Konzept (Bildquelle: [TM15])

Ein Extension Point ermöglicht die Integration eines eigenen *Composers*, der aus den verschiedenen *solution space*-Artefakten einzelne Produkte zusammensetzt. Der *Composer* leitet aus dem Softwareproduktlinienmodell und der jeweiligen Feature-Konfiguration konkrete Produktarchitekturen ab.

6.3 Ein Überblick über den Prototypen

Der entwickelte Prototyp wird aus zwei verschiedenen Blickwinkeln beschrieben. Zum einen wird die innere Struktur des Datenmodells und der Software erläutert. Sowohl die Daten als auch die Funktionalität wird in kleinere Module dekomponiert. Der zweite Blickwinkel verdeutlicht, wie mit der Software ein Softwareproduktlinienmodell erstellt werden kann und wie daraus eine Produktarchitektur abgeleitet wird.

6.3.1 Modularisierung des Systems

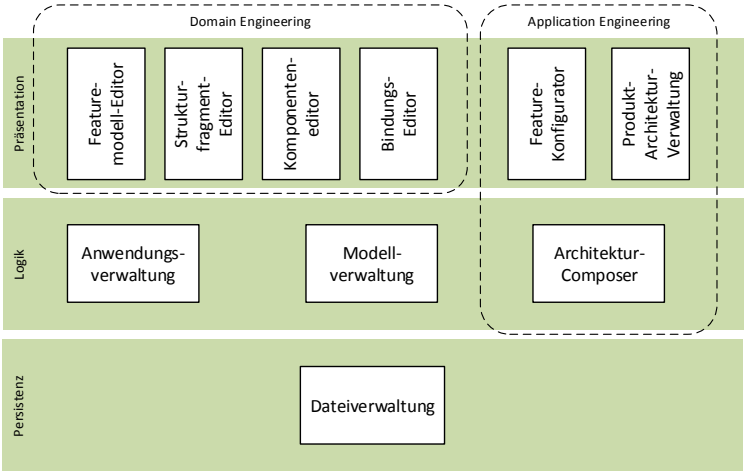


Abbildung 6.5: Notwendige Zerlegung des Systems in fachliche Module

Für eine grobe Dekomposition des Systems in einzelne fachliche Module wird die Zerlegung des Entwicklungsprozesses (s. Abbildung 4.2) als Vorbild genommen. Für jeden einzelnen Schritt wird ein eigenes Modul vorgesehen, mit dem beispielsweise die entsprechenden Artefakte erstellt werden können. An der gestrichelten Umrandung in Abbildung 6.5 ist die Zuordnung zu einem der Hauptschritte (*domain engineering* und *application engineering*) des Prozesses dargestellt.
Die Forderung, die Daten in verschiedenen Dateien zu speichern, erfordert weitere Module. Es wird eine Dateiverwaltung benötigt, die zum einen für das Schreiben und Einlesen der entsprechenden Dateien zuständig ist, zum anderen aber auch verwaltet, welche Dateien zur Softwareproduktlinie gehören.
Für die Ableitung einer konkreten Architektur müssen diese Datenmodule zu einem Gesamt-Softwareproduktlinienmodell zusammengefügt werden. Diese Aufgabe übernimmt das Modul Modellverwaltung. Das Zusammenspiel der Softwaremodule wird durch eine Anwendungsverwaltung realisiert.

Die Abbildung 6.5 zeigt die notwendige Zerlegung in einzelne Module. Die verschiedenen Module werden auf drei Ebenen verteilt. Die Präsentations-Ebene enthält die Module, die über eine grafische Benutzerschnittstelle verfügen. Die Anwendungs- und Modellverwaltung werden mit dem Architektur-Composer der Logik-Ebene zugeordnet. Die Dateiverwaltung ist ein Modul der Persistenz-Ebene.

Der Feature-Modell-Editor ist Bestandteil des FeatureIDE-Projekts. Mit ihm lässt sich ein Feature-Modell erstellen, das die Eigenschaften aufweist, die das Metamodell aus 5 fordert. [Thü+14]

Zur Modellierung der *solution space*-Artefakte kommen zwei GMF-basierte Editoren zum Einsatz. Mit dem Strukturfragmente-Editor lässt sich ein Strukturfragment bearbeiten. Der Komponenten-Editor ist zur Beschreibung einer Komponente integriert. Beide können jeweils genau ein Artefakt bearbeiten, das jeweils in einer eigenen Datei gespeichert wird. Mit dem Bindungs-Editor lassen sich die Artfakte zu Feature-Implementierungen verbinden.

Das *application engineering* wird mit zwei Editoren unterstützt. Der Feature-Konfigurator ermöglicht die Ableitung valider Feature-Konfigurationen. Dieser ist Bestandteil des FeatureIDE-Projekts. Über einen Extension Point kann die Funktion durch einen so genannten Architektur-Composer erweitert werden, der für die automatisierte Ableitung von Produktarchitekturen zuständig ist. Die Ergebnisse können mit der Produkt-Architektur-Verwaltung untersucht und bewertet werden.

Da das Softwareproduktlinienmodell auf mehrere Dateien aufgeteilt ist, muss es eine Komponente geben, die das Softwareproduktlinienmodell als Ganzes verwaltet und benötigte Informationen anderen Komponenten wie dem Bindungs-Editor oder dem Architektur-Composer zur Verfügung stellt. Andere notwendige Aufgaben, die zur Anwendungsverwaltung oder Dateiverwaltung gehören wie zum Beispiel das Viewmanagement werden komplett an Eclipse Komponenten delegiert.

6.3.2 Strukturierung der Daten

Die Grundstruktur des Datenmodells folgt aus der Struktur des Metamodells (s. Kapitel 5). Das Modell der Softwareproduktlinie liegt in mehreren Dateien vor. Für eine geeignete Zerlegung des Modells werden die Paketgrenzen, die in Abbildung 5.2 dargestellt sind, berücksichtigt. Das Architekturmodell wird noch weiter zerlegt, sodass in einer Datei genau ein Strukturfragment beschrieben ist. Auch in der Komponentenbibliothek wird für jede Komponentendefinition eine eigene Datei vorgesehen.

Dadurch wird die Zusammenführung (*Merge*) der Teile aus den Dateien zu einem Gesamtmodell deutlich vereinfacht. Im Prototypen ist ein verhältnismäßig einfaches Verfahren für einen *Merge* der Teilmodelle nutzbar, da die zu *mergenden* Teilmodelle disjunkt zueinander sind.
Alle Komponenten werden zur Komponenten-Bibliothek und alle Strukturfragmente zum Architekturmodell hinzugefügt. Die einzigen Konflikte, die dabei entstehen können, sind einfach zu entdecken. Es handelt sich dabei um identische Belegungen bei verschiedenen im Metamodell mit `{unique}` gekennzeichneten Attributen.

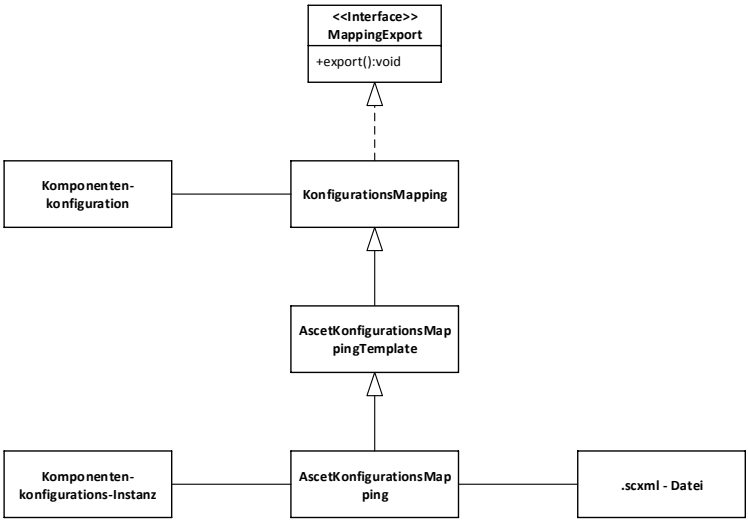


Abbildung 6.6: Die Erweiterung des Datenmodells um die für die Modelltransformation in ein PSM benötigten Informationen am Beispiel einer Komponenten-Konfiguration

Um mit dem Prototypen konkrete Produkte erzeugen zu können, soll nicht nur eine abstrakte Beschreibung der Architektur ausgegeben werden. Vielmehr sollen auch gleich die entsprechenden Software-Artefakte inklusive entsprechender Parametrierung für eine Weiterverarbeitung zur Verfügung stehen. Im Sinne einer modellgetriebenen Softwareentwicklung (s. MDA in Kapitel 2) repräsentieren die modellierten Daten ein plattformunabhängiges Modell (PIM) der Softwareproduktlinie. Das bedeutet,

dass eine abgeleitete Produktarchitektur ein plattformunabhängiges Modell eines Produktes ist. Mit einer Modelltransformation kann daraus ein plattformspezifisches Modell generiert werden.

Dieser Teil des Prototypen liegt außerhalb des betrachteten Fokus dieser Arbeit. Dennoch soll an dieser Stelle ein Hinweis darauf gegeben werden, was das Datenmodell zusätzlich leisten muss, um eine Transformation zu ermöglichen. Im vorliegenden Beispiel ist eine entsprechende Erweiterung relativ einfach zu realisieren.

Das Datenmodell wird um die Möglichkeit zur Speicherung zusätzlicher Informationen erweitert. Der Prototyp muss nach der Auswahl der gewünschten Produktarchitektur im *application engineering* die Zusatzinformationen der Architekturelemente ausgeben.

Abbildung 6.6 zeigt am Beispiel einer Komponenten-Konfiguration, wie sich diese Anforderung auf das Datenmodell des Prototypen auswirkt. Für jedes relevante Element im Softwareproduktlinienmodell ist ein Mapping auf ein entsprechendes Software-Artefakt definiert worden. Für die Ausgabe der gespeicherten Informationen ruft der Prototyp die Methode `export()` auf. Hierfür müssen alle Mapping-Klassen das entsprechende Interface realisieren.

Die Spezifikation des Mappings erfolgt in zwei Ebenen. Zuerst wird abhängig vom Modell-Element-Typ ein Mapping-Template spezifiziert. Dieses Mapping-Template beschreibt den grundsätzlichen Aufbau der Klasse zur Speicherung entsprechender Informationen. Das Mapping-Template gibt somit gewissermaßen die abstrakte Syntax der plattformspezifischen Modellanteile vor.

Wenn die Variabilität in Komponenten durch `#ifdef`-Präprozessordirektiven realisiert wird, könnte eine solche Mapping-Template-Klasse eine Variable zum Speichern eines Dateipfades enthalten. Die Implementierung der `export()`-Methode könnte das Kopieren dieser Datei in ein bestimmtes Verzeichnis realisieren.

In der zweiten Ebene der Mapping-Spezifikation muss der Inhalt des Templates angegeben werden. Im zuvor genannten Beispiel wird der Pfad auf eine konkrete Datei mit entsprechenden Präprozessor-Makros eingetragen. Beim *build*-Prozess wird darüber die entsprechende Variante erzeugt.

Der Prototyp stellt für eine Erweiterung der Daten Klassen zur Verfügung, die um projektspezifische Informationen zu den jeweiligen Templates erweitert werden können. Folgende Modell-Elemente können auf diese Weise verfeinert werden: `Eingangs-Port`, `Ausgangs-Port`, `Port-Typ`, `Komponente`, `Komponenten-Konfiguration` und `Link`.

6.3.3 Erstellung eines Produktlinienmodells

Abbildung 6.7 zeigt einen Screenshot des Tools[1]. In diesem sind die Editoren zu sehen, die für die Modellierung im *domain engineering* Schritt vorgesehen sind. Auf der linken Seite des Bildes ist ein Projekt-Explorer dargestellt. In einem Projekt wird genau eine Softwareproduktlinie verwaltet. Da diese auf verschiedene Dateien verteilt ist, werden die entsprechenden Artefakte in einer Projektstruktur verwaltet, die in Abbildung 6.8 detailliert dargestellt ist.

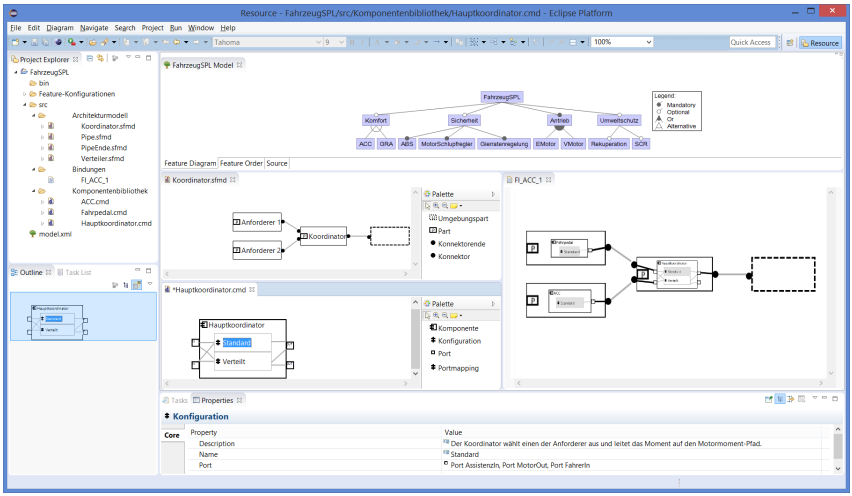


Abbildung 6.7: Screenshot der grafischen Oberfläche mit den Editoren für das *domain engineering*

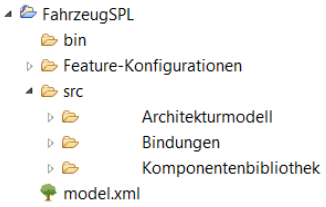


Abbildung 6.8: Screenshot der Projektstruktur

[1]Der Editor für Feature-Implementierungen ist durch ein Mockup ersetzt.

Mit Hilfe eines *Wizards* können neue Artefakte in das Projekt integriert werden. Das Beispiel in Abbildung 6.9 zeigt exemplarisch den Wizard, mit dem eine neue Komponente angelegt werden kann. Eine entsprechende Datei wird erzeugt, an der richtigen Stelle im Projekt einsortiert, und der jeweilige Editor automatisch geöffnet.

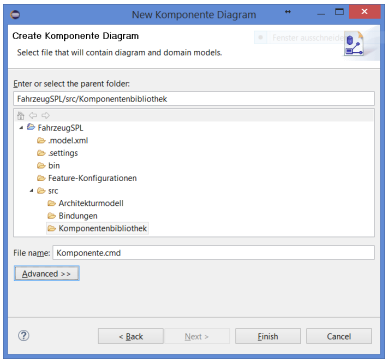


Abbildung 6.9: Screenshot des Wizards zum Anlegen einer neuen Komponente

Die Feature-Modelle, die mit dem FeatureIDE-Editor erstellt werden können, erfüllen alle Anforderungen, die ein Feature-Modell laut Kapitel 5 leisten muss. Features sind monohierarchisch angeordnet. Sie können als optional oder verpflichtend deklariert werden. Gruppierungen in „ODER“- beziehungsweise „ALTERNATIV“-Gruppen sind ebenfalls möglich. Darüber hinaus lassen sich Querbeziehungen von beliebigen Features mit den Eigenschaften „benötigt“ oder „verbietet“ herstellen.
Die grafische Oberfläche besteht nur aus einer Diagramm-View. Abbildung 6.10 zeigt ein im Editor dargestelltes Modell. Die konkrete Syntax des Modells entspricht weitgehend der in Abschnitt 5.3 spezifizierten.
Die wesentlichen Unterschiede sind zum einen die Vertauschung der Symbole für optionale und verpflichtende Features. Ein ausgefüllter Kreis an der oberen Kante eines Features bedeutet beispielsweise, dass das Feature im Gegensatz zur Spezifikation in Kapitel 5 nicht optional sondern verpflichtend in jeder Feature-Konfiguration zu verwenden ist.
Zum anderen sind die Querbeziehungen nicht als gestrichelte Kanten im Diagramm zwischen den beteiligten Features dargestellt. Stattdessen werden sie als einfache logische Ausdrücke unter dem Diagramm rein textuell aufgelistet. Beispielsweise wird die Querbeziehung, dass das VMotor-Feature in der Feature-Konfiguration enthalten sein muss, wenn

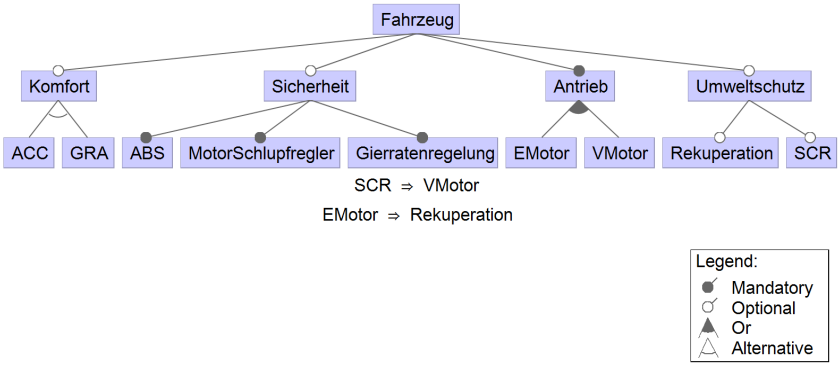


Abbildung 6.10: Ein Feature-Modell dargestellt in der Diagrammfläche des *FeatureIDE* Editors

das SCR-Feature gewählt wurde, durch die folgende Implikation ausgedrückt: $SRC \Rightarrow VMotor$.

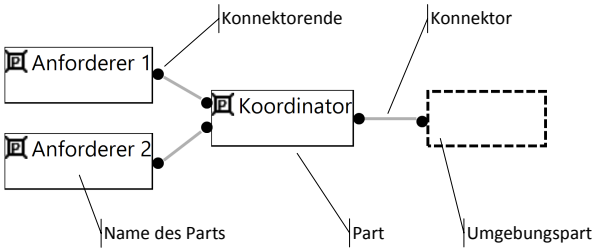


Abbildung 6.11: Darstellung eines Strukturfragments in der Diagrammfläche des Editors

In der Mitte der Abbildung 6.7 ist die View des Strukturfragment-Editors dargestellt. Eine Detailaufnahme der Diagrammfläche zeigt Abbildung 6.11. Die einzelnen Elemente können über die Palette oder das Kontextmenü erzeugt und miteinander verbunden werden.

Auf die gleiche Weise lassen sich Komponenten in der Diagrammsicht des Komponenten-Editors erstellen. Das Gesamtbild (Abbildung 6.7) zeigt die entsprechende View unten in der Mitte. Der entsprechende Ausschnitt ist in Abbildung 6.12 vergrößert dargestellt.

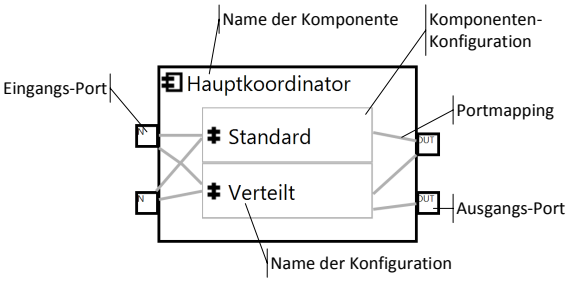


Abbildung 6.12: Darstellung einer Komponente im Komponenten-Editor

Beim Speichern des editierten Artefakts wird automatisch im Hintergrund die Modell-Verwaltung aktiviert, um das verwaltete Softwareproduktlinienmodell zu aktualisieren. Dieser Schritt ist zwingend notwendig, da bei der Erstellung von Feature-Implementierungen natürlich alle Artefakte, die miteinander verbunden werden sollen, bekannt sein müssen.
Die Feature-Implementierungen selbst werden ebenfalls als einzelne Dateien erstellt. Das Diagramm, das der Editor für die Erstellung nutzt, entspricht der Modellierungssicht auf Feature-Implementierungen. Abbildung 6.13 zeigt einen detaillierteren Ausschnitt dieses Diagramms als Mockup. Die ausgewählte Konfiguration wird dabei, wie in Abschnitt 5.1.4 vorgeschlagen, farblich hervorgehoben.

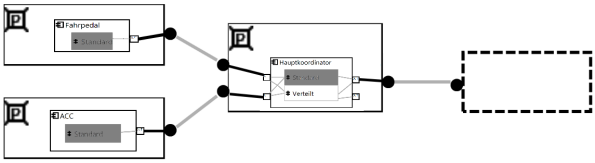


Abbildung 6.13: Darstellung einer Feature-Implementierung im Bindungseditor

6.3.4 Ableitung einer Architektur

Der Screenshot in Abbildung 6.14 zeigt die grafische Oberfläche des Prototypen beim *application engineering*[2]. Dargestellt ist eine mögliche Anordnung

[2]Die View zur Darstellung der Produktarchitektur ist durch ein Mockup ersetzt.

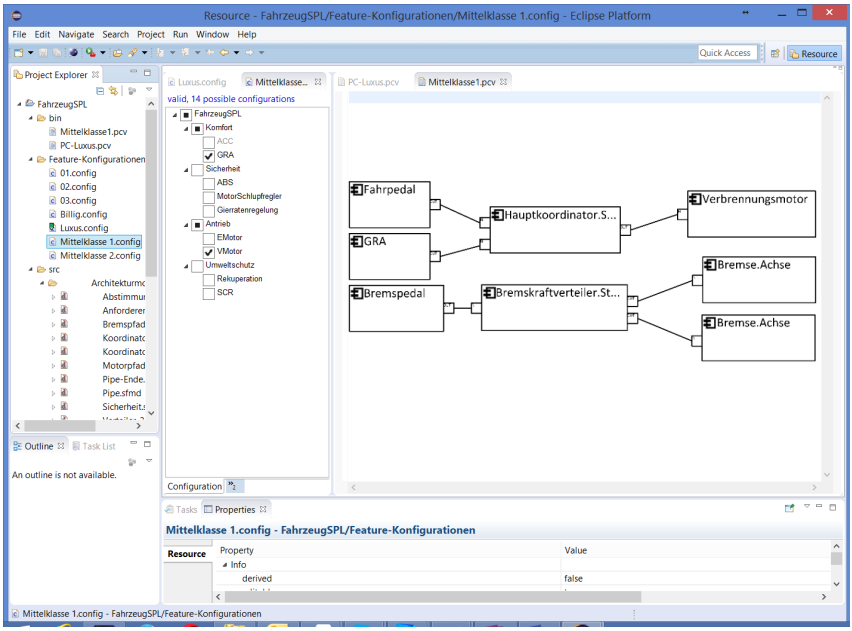


Abbildung 6.14: Screenshot der grafischen Oberfläche mit den Editoren für das *application engineering*

der relevanten Editoren. In der Mitte befindet sich der Konfigurationseditor. Dieser ist in einem detaillierteren Ausschnitt in Abbildung 6.15 zu sehen. Der Konfigurationseditor ist Bestandteil von *FeatureIDE*.

Für die Präsentation wird eine *treeview* genutzt, in der die Featurenamen entsprechend der Baumstruktur aus dem Feature-Modell einsortiert sind. Jeder Name ist mit einer Checkbox versehen. Der Architekt selektiert die Features, die in der Konfiguration enthalten sein sollen. Der Editor verändert die Feature-Konfiguration selbständig, wenn eine entsprechende Randbedingung (Querbeziehung oder Feature-Gruppeneigenschaft) das fordert.

Erfordert die Auswahl eines Features die Existenz eines anderen in der Feature-Konfiguration, wird das zusätzliche Feature in der Feature-Konfiguration automatisch aufgenommen. Die Checkbox wird in dem Fall mit einem schwarzen Quadrat gefüllt. Verbietet die Feature-Auswahl die Existenz eines anderen Features, wird der entsprechende Name mit grauer Schrift geschrieben und die Checkbox deaktiviert (vgl. ACC und GRA in

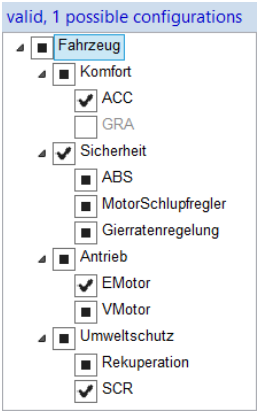


Abbildung 6.15: Screenshot des Feature-Konfigurations-Editors

Abbildung 6.15). Dadurch wird eine invalide Konfiguration durch Verletzung der entsprechenden Randbedingungen verhindert.

Es ist allerdings noch möglich, eine invalide Konfiguration durch Unterspezifikation zu erzeugen. Im Beispiel wäre das der Fall, wenn zum Beispiel weder ACC noch GRA ausgewählt wäre. Der Editor erkennt das und zeigt oberhalb der *treeview* an, ob die aktuelle Feature-Konfiguration valide ist oder nicht.

Bei Speicherung der Konfiguration wird über den *Extension-Point* im FeatureIDE der Architektur-Composer automatisch gestartet. Dieser erzeugt im ersten Schritt eine Menge von möglichen Produkt-Architekturen. Dafür ist die in Abschnitt 4.4 beschriebene Methode zur Ableitung von Architekturen aus dem SPL-Modell umgesetzt. Der Architekt kann die ungültigen Ergebnisse markieren. Diese werden bei der erneuten Konfiguration derselben Features nicht mehr als potentielle Lösung vorgeschlagen. Aus den gültigen Lösungen wählt der Architekt die aus, die den technischen Randbedingungen genügt, die einen direkten Einfluss auf den *solution space* haben. Die Architektur repräsentiert eine Produktvariante, die sowohl die fachlichen Aspekte des *problem space* wie auch die technisches Aspekte des Kontextes in der Domäne berücksichtigt (vgl. Abschnitt 3.2.2).

Der Architektur-Composer führt in einem zweiten Verarbeitungsschritt die Transformation in ein plattformspezifisches Modell durch. Bei der oben erwähnten Implementierung werden die `export()`-Methoden der entsprechenden Architekturelemente aufgerufen. Die Ergebnisse sind die

für die weitere Verarbeitung benötigten Software-Artefakte, die in einem speziellen Verzeichnis abgelegt werden.

6.4 Zusammenfassung

In diesem Kapitel wird beschrieben, wie mit einer prototypischen Umsetzung eines Tools auf Basis des fragmentbasierten Ansatzes eine featureorientierte Softwareproduktlinie modelliert werden kann. Am Anfang des Kapitels wird ein Überblick über die wesentlichen Anforderungen an das Werkzeug und ausgewählte technische Randbedingungen gegeben. Im Anschluss daran folgt ein sehr abstrakter Einblick in die System-Struktur und eine Beschreibung der notwendigen Erweiterungen des Datenmodells. Den Schwerpunkt bildet allerdings die Beschreibung des Prototypen aus Sicht der Architekten, die mit dem Tool arbeiten. Da nicht alle Teile der Software im Rahmen dieser Dissertation umgesetzt werden konnten, werden für die Beschreibung sowohl Screenshots erstellter Editoren als auch Mockups genutzt. Die Präsentation erfolgt in zwei Teilen. Der erste zeigt die Möglichkeiten zur Erstellung eines Softwareproduktlinienmodells. Der zweite Teil konzentriert sich auf die Ableitung einer konkreten Produktarchitektur.
Auf eine komplette Spezifikation oder Dokumentation des Prototypen wird bewusst verzichtet, da dies das Ziel dieses Kapitels deutlich überschreiten würde. Für die Vermittlung der Idee, wie das Konzept in einem Prototypen umgesetzt werden kann, ist der gewählte Abstraktionsgrad hinreichend.

Kapitel 7

Fallstudie

In der Motivation für diese Dissertation, die zur Forschungsfrage führte (s. Kapitel 3.3), ist der Wunsch genannt, dass Architekten bei Ihrer Arbeit mit Produktlinien unterstützt werden sollen. Obwohl über verschiedene Projekte ein enger Kontakt zu Architekten von Volkswagen existierte, war es leider nicht möglich, eine umfassende Evaluierung des Konzepts durchzuführen.
Stattdessen wird anhand des Beispiels aus Kapitel 3.1 die grundsätzliche Anwendbarkeit demonstriert. Der erste Teil des Kapitels beschreibt das *domain engineering*. Es wird darin gezeigt, welche Modellelemente entwickelt und wie diese miteinander verbunden wurden. Der zweite Teil befasst sich mit dem *application engineering*. Es werden exemplarisch Ableitungen von Produktarchitekturen präsentiert. Im letzten Abschnitt werden einige Erkenntnisse genannt, die bei der Anwendung des Konzepts gewonnen wurden.

7.1 Modellierung der Beispiel-Softwareproduktlinie

Die gezeigten Beispiel-Architekturen in Kapitel 3.1 repräsentieren eine sehr abstrakte, rein logische Sicht auf die jeweiligen Systeme. In dieser Sicht sind nicht alle Varianten erkennbar, die in den Softwarekomponenten enthalten sind. Der Grund liegt in den technischen Randbedingungen, die durch die Wahl der Modellierungssprache impliziert sind. Die Komponenten im Beispiel-Projekt liegen als Modelle vor, die in der Modellierungssprache *ASCET*[1] [Gmb07] spezifiziert sind. Diese projektspezifischen Randbedingungen werden im Folgenden erwähnt.
In diesem Abschnitt wird zuerst noch einmal das modellierte Featuremodell präsentiert. Darauf folgt eine Beschreibung des Architekturmodells und

[1] ASCET ist eine Entwicklung der ETAS GmbH, Stuttgart [16a]

der Komponentenbibliothek. Den Abschluss bildet eine Auflistung der Bindungselemente.

7.1.1 Featuremodell

Das Featuremodell des Beispiels wurde in der Arbeit schon mehrfach gezeigt. Natürlich wurde es bei der Fallstudie genauso angelegt, wie es das Beispiel fordert. Der Vollständigkeit halber wird es an dieser Stelle noch einmal gezeigt (s. Abbildung 6.10).

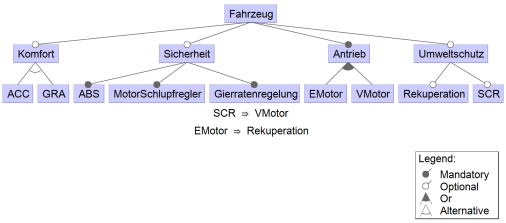


Abbildung 7.1: Das Featuremodell des Beispielprojekts, modelliert mit dem Prototypen

7.1.2 Architekturmodell

Das Architekturmodell besteht aus einer überschaubaren Menge an Strukturfragmenten. In Tabelle 7.1 sind alle mit einer kurzen Erklärung aufgelistet. Bei der Modellierung zeigte sich, dass sich die Module je nach Einsatzzweck in ihrer Komplexität deutlich voneinander unterscheiden können.

Abbildung 7.2: Beispiel eines Strukturfragments für die Darstellung des Pipes-And-Filters-Musters

Wird ein Strukturfragment entworfen, um ein Entwicklungsmuster zu realisieren, lässt es sich für die Modellierung verschiedener Architekturteile sowohl innerhalb einer Produktarchitektur als auch über Produktgrenzen hinaus wiederverwenden. Ein solches Strukturfragment ist beispielsweise das `pipe`-Strukurfragment (s. Abbildung 7.2) Dabei ist die Flexibilität bezüglich der modellierbaren Systemstruktur-Variabilität sehr groß, denn

es gibt entsprechend viele Möglichkeiten, die Strukturfragmente zu verschmelzen. Der Modellierungsaufwand ist jedoch gering.

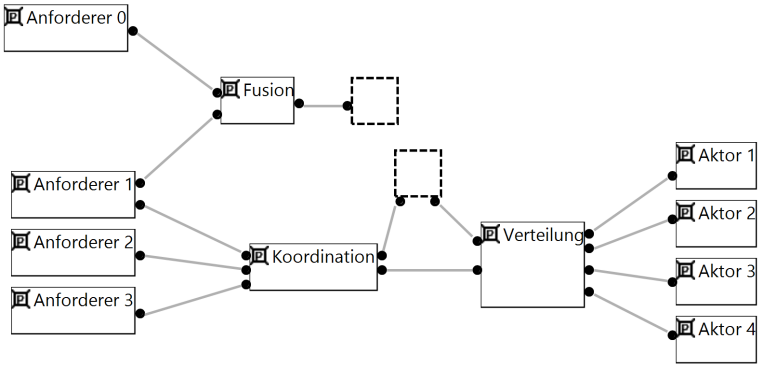


Abbildung 7.3: Das Strukturfragment, das nur für die Implementierung des Features `Sicherheit` eingesetzt wird

Das Strukturfragement `Sicherheit` repräsentiert dahingegen eine eher komplexe Lösung (s. Abbildung 7.3). Diese wurde explizit für die Beschreibung einer sehr produktspezifischen Strukturlösung erschaffen. Die Systemstruktur-Variabilität ist in dem entsprechenden Ausschnitt der Produktarchitekturen sehr eingeschränkt. Das zeigt, dass sich Strukturfragmente gut eignen, um Variabilität hinsichtlich der Systemstruktur modellieren zu können ohne gleichzeitig die Anzahl der Strukturfragmente zu stark wachsen zu lassen.

Nr.	Name	Beschreibung
1	Abgas	Das Abgas-Fragment ist speziell zum Aufbau des Abgasstrangs geeignet.
2	Abstimmung	Das Abstimmung-Fragment ist eine Spezialstruktur, die nur zum Aufbau einer Bremslichtabstimmung bei Fahrzeugen mit Elektromotor dient.
3	Bremsenpfad	Repräsentiert die spezielle Minimalstruktur, die ein Fahrzeug als Bremsenpfadstruktur nutzen kann.
4	Koordination	Das Koordination-Strukturfragment ermöglicht die Zusammenführung mehrerer Datenströme.
5	Pipe	Das Pipe-Fragment erlaubt die serielle Verschaltung von Komponenten.

Nr.	Name	Beschreibung
6	Sicherheit	Das Sicherheit-Strukturfragment ist eine komplexe Spezialstruktur, die nur für den Aufbau eines Bremsenpfads mit Sicherheitsfeatures geschaffen ist.

Tabelle 7.1: Liste der modellierten Strukturfragmente

7.1.3 Komponentenbibliothek

Die Komponenten, die für die Erstellung der Beispielproduktlinie modelliert wurden, werden in Tabelle 7.2 aufgelistet. Prinzipiell erlaubt auch eine in ASCET modellierte Komponente, dass sie durch eine andere im System ausgetauscht werden kann, wenn diese über kompatible Schnittstellen verfügt. Die Komponentenvariabilität wurde entsprechend durch Definition verschiedener Komponenten mit gleichartigen Ports vom selben Typen realisiert. Ein Beispiel hierfür sind die Komponenten `GRA` und `ACC`. Beide Komponenten bieten einen Ausgangsport vom gleichen Typen an.
Die ASCET-Komponenten bieten durch gleiche Portspezifikationen nicht nur die Möglichkeit zur Erstellung von Komponentenvariabilität. Es gibt auch eine Möglichkeit, die Konfigurationsvariabilität zu modellieren. Eine konkrete Variante wird durch Angabe einer speziellen Parametrierung selektiert.

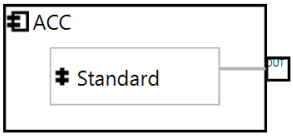


Abbildung 7.4: ACC ist eine Komponente, die über nur eine Konfiguration verfügt

Ein Großteil der Komponenten ist so „einfach“ gestaltet, dass sie in den verschiedenen Umgebungen eingesetzt werden können, ohne dass mehrere Komponentenkonfigurationen erforderlich sind. Auch hierfür ist die in Abbildung 7.4 dargestellte Komponente `ACC` ein Beispiel. Sie enthält nur eine Konfiguration, weil die Funktionalität in jedem Produkt identisch ist. Andere Komponenten verfügen über mehrere Konfigurationen. Die Anzahl der Konfigurationen hängt von verschiedenen Entscheidungen ab. Die `Bremslichtansteuerung` verfügt über eine Funktionalität, die über verschiedene Ports zur Verfügung gestellt wird. Beim `Hauptkoordinator` sind

mehrere ähnliche Funktionalitäten, die unterschiedliche Ports benötigen, in einer Komponente gekapselt. Der Grund für mehrere Konfigurationen der Komponente `DP-Filter` ist jedoch ausschließlich auf technische Randbedingungen zurückzuführen. Die Anordnung dieser Komponenten zueinander muss variieren können.
An den Komponenten `Bremskraftverteiler` und `Bremskraftverteiler-ESP` lässt sich erkennen, dass es unterschiedliche Realisierungsmöglichkeiten für die Variabilität gibt. Es wurde festgestellt, dass es durchaus sinnvoll sein kann, die Variabilitätsformen zu kombinieren. Statt die Variabilität an dieser Stelle ausschließlich durch Konfigurationsvariabilität zu realisieren, wurde zusätzlich ein Teil der Variabilität auf Komponentenvariabilität abgebildet.
Der Sinn dieser Verteilung zeigt sich bei der Erstellung der Feature-Implementierungen. Die ESP-Funktionalität kann nur an das Sicherheits-Strukturfragment gebunden werden. Umgekehrt darf die Standard-Funktionalität der Bremsverteilung genau bei diesem Strukturfragment nicht zum Einsatz kommen. Durch Aufteilung auf zwei Komponenten konnte die Komplexität der einzelnen Komponenten verringert werden, ohne die Komplexität in anderen Teilen (Bibliothek oder Bindungsmenge) in gleichem Maße zu erhöhen.

Nr.	**Name**	$\#_{Knf}$	**Beschreibung**
1	ABS	1	Das Anti-Blockiersystem benötigt nur eine Standardkonfiguration.
2	ACC	1	Das ACC benötigt ebenfalls nur eine Konfiguration.
3	Bremse	2	Die Komponente steuert entweder die Bremse eines einzelnen Rades oder die Bremsen aller Räder einer Achse an.
4	Bremskraftverteiler	4	Die Bremskraftverteiler-Komponente für einfache Bremssysteme kann zwei unabhängige zusätzliche Funktionen enthalten: Koordination zweier Anforderer oder eine Abstimmung bei Rekuperations-Systemen.
5	Bremskraftverteiler-ESP	4	Die gleiche Konfigurationsvariabilität bietet der Bremskraftverteiler mit ESP-Funktionalität als Basisfunktion.
6	Bremslichtansteuerung	2	Die Bremslichtansteuerung wird entweder von einem oder zwei Anforderern angesteuert.

Nr.	Name	$\#_{Knf}$	Beschreibung
7	Bremspedal	1	Die Bremspedalkomponente verfügt über keine innere Variabilität. Die einzige Konfiguration liest den Pedalwinkel aus.
8	Dosierungs-Steuerung	1	Die Dosierungs-Steuerung benötigt nur eine Konfiguration. Das Attribut `minLinks` am Ausgangs-Port ist mit 0 belegt. Darum kann die Komponente auch am Ende der Verarbeitungskette eingesetzt werden.
9	DP-Filter	1	Der Dieselpartikelfilter benötigt nur eine Konfiguration. Das Attribut `minLinks` am Ausgangs-Port ist mit 0 belegt. Darum kann die Komponente auch am Ende der Verarbeitungskette eingesetzt werden.
10	Elektromotor	1	Die Elektromotor-Komponente liegt nur in einer Variante ohne Konfigurationsvariabilität vor.
11	Fahrermoment	1	Die Fahrermoment-Komponente berechnet immer das Gesamt-Wunschmoment des Fahrers auf gleiche Weise aus den Pedalwinkeln, weshalb nur eine Konfiguration notwendig ist.
12	Fahrpedal	1	Die Fahrpedal-Komponente hat wie die Bremspedal-Komponente nur eine Konfiguration zur Erfassung des Pedalwinkels.
13	GRA	1	Das GRA unterscheidet sich vom ACC in der Architektur nur in der inneren Funktionalität, die Ports sind vom gleichen Typ.
14	Hauptkoordinator	6	Durch unterschiedliche Kombinationen von Eingangs- und Ausgangsportnutzung enthält der Hauptkoordinator sechs verschiedene Konfigurationen.
15	Motorschlupfregler	1	Diese Komponente hat nur eine Standardkonfiguration zur Berechnung des durch den Motor bewirkten Schlupf an den Rädern.

Nr.	Name	$\#_{Knf}$	Beschreibung
16	Oxi-Kat	1	Die Oxidationskatalysator-Komponente hat nur eine Konfiguration, da ihr Verhalten immer gleich ist und sich ihre Position in der Komposition nicht ändert.
17	Verbrennungsmotor	4	Der Verbrennungsmotor verfügt über weitere Konfigurationen für die Bereitstellung eines Rekuperationsports und eines Abgasports.
18	Vorkoordinator	1	Auch der Vorkoordinator wird immer in der gleichen Variante verbaut. Er koodiniert die Momenteingriffe auf Seite des ESP-Steuergeräts.

Tabelle 7.2: Liste der modellierten Komponenten mit Angabe der Anzahl enthaltener Konfigurationen ($\#_{Knf}$)

7.1.4 Bindungsmodell

Der letzte Schritt des *domain engineering*s ist die Erstellung der Bindungen zwischen den verschiedenen erstellten Modellen. Diese Bindungen vereinigen die in den vorherigen Abschnitten erstellten Modelle zu einem Softwareproduktlinienmodell. Die Menge der notwendigen Feature-Implementierungen ist in Tabelle 7.3 aufgelistet.
Jede Zeile in dieser Tabelle beschreibt, welche Teile in einer Feature-Implementierung gebunden werden. An welche Parts des jeweiligen Strukturfragments die Komponenten gebunden werden, zeigt die Tabelle allerdings nicht. Zu erkennen ist, dass sowohl Strukturfragmente als auch Komponenten für verschiedene Feature-Implementierungen wiederverwendet werden. Als Beispiel sei das Strukturfragment `Koordination` genannt, das sowohl für die Implementierung der Features ACC als auch GRA genutzt wird. Der Verbrennungsmotor ist ein Beispiel für eine in verschiedenen Feature-Implementierungen genutzte Komponente.

Nr.	Features	Strukturfragment	Komponenten/-konfigurationen
1	Fahrzeug	Bremsenpfad	Bremse.Achse, Bremse.Achse, Bremskraftverteiler.Standard

Nr.	Features	Struktur-fragment	Komponenten/-konfigurationen
2	Fahrzeug, Sicherheit	Sicherheit	Bremse.Rad, Bremse.Rad, Bremse.Rad, Bremse.Rad, Bremskraftverteiler-ESP, Vorkoordinator, Motorschlupfregler, Bremspedal, ABS, Bremslichtansteuerung, Fahrermoment, Fahrpedal, Hauptkoordinator
3	ACC	Koordination	ACC, Fahrpedal, Hauptkoordinator
4			ACC, Fahrermoment, Hauptkoordinator
5	GRA, Komfort	Koordination	GRA, Fahrpedal, Hauptkoordinator
6			GRA, Fahrermoment, Hauptkoordinator
7	EMotor	Abstimmung	Elektromotor, Bremskraftverteiler, Bremslichtansteuerung
8			Elektromotor, Bremskraftverteiler-ESP, Bremslichtansteuerung
9	Antrieb, Rekuperation	Pipe	Hauptkoordinator, Elektromotor
10		Pipe	Fahrermoment, Elektromotor
11			Fahrpedal, Elektromotor
12	Antrieb, VMotor	Pipe	Hauptkoordinator, Verbrennungsmotor
13			Fahrermoment, Verbrennungsmotor
14			Fahrpedal, Verbrennungsmotor
15	Rekuperation	Pipe	Hauptkoordinator, Verbrennungsmotor.Reku

Nr.	Features	Struktur-fragment	Komponenten/-konfigurationen
16			Fahrermoment, Verbrennungsmotor.Reku
17			Fahrpedal, Verbrennungsmotor.Reku
18	SCR	Abgas	Verbrennungsmotor, Oxi-Kat, DP-Filter.Vorn, Dosierungssteuerung.Hinten
19			Verbrennungsmotor, Oxi-Kat, DP-Filter.Hinten, Dosierungssteuerung.Vorn
20	Komfort	Pipe	Hauptkoordinator, Bremskraftverteiler
21			Hauptkoordinator, Bremskraftverteiler-ESP

Tabelle 7.3: Liste der modellierten Feature-Implementierungen mit Angabe der referenzierten Artefakte

Einige der Feature-Implementierungen realisieren mehrere Features. Außerdem werden einige Features durch unterschiedliche Feature-Implementierungen realisiert. Das bedeutet, dass das in Kapitel 3 gewünschte n-zu-m-Mapping zwischen Elementen des *problem space* und Elementen des *solution space* möglich ist.

7.2 Ableitung von Produkten

Das *application engineering* soll zuerst an einem einfachen Produkt demonstriert werden. Das ausgewählte Produkt ist die Variante „Billig“. Bei der entsprechenden Feature-Konfiguration ist tatsächlich nur eine Lösung möglich.
Aufgrund der Einfachheit dieses Produkts können die einzelnen Phasen der Architekturableitung (s. Abbildung 4.15) ausführlicher behandelt werden. Es werden die einzelnen Zwischenergebnisse, die nach der entsprechenden Phase vorliegen, gezeigt. Im Anschluss daran wird an einem weiteren Beispiel gezeigt, wie alternative Lösungen für komplexere Produkte nach der Vorgabe einer Feature-Konfiguration entstehen können.

Die Produktarchitektur der Variante „Billig“, die automatisiert abgeleitet werden soll, muss die Struktur haben, die in Abbildung 3.7 dargestellt ist. Die entsprechende Feature-Konfiguration ist sehr einfach. In Abbildung 7.5 ist zu erkennen, dass nur ein Feature ausgewählt ist.

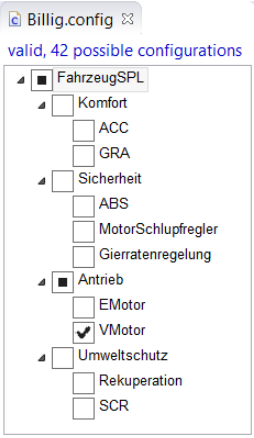


Abbildung 7.5: Feature-Konfiguration der Variante „Billig“

Die erste Phase der Architekturableitung ist die Filterung der Feature-Implementierungen. Aus Tabelle 7.3 können die Nummern der Feature-Implementierungen entnommen werden, die die Features der Feature-Konfiguration grundsätzlich realisieren können. Die Menge dieser Nummern ist nach dem ersten Schritt {1, 9, 10, 11, 12, 13, 14}. Dabei ist zu bedenken, dass die meisten Feature-Implementierungen jeweils zwei Features realisieren. Da eins der beiden Features bei den Feature-Implementierungen 9, 10 und 11 jeweils `Rekuperation` ist, werden diese drei Artefakte ebenfalls herausgefiltert. Das Feature `Rekuperation` ist nicht in der Feature-Konfiguration enthalten. Nach Abschluss der ersten Phase besteht die Menge der Feature-Implementierungen aus folgenden Elementen: {1, 12, 13, 14}.
In der zweiten Phase werden diese Feature-Implementierungen so gruppiert, dass in jeder Menge jedes Feature nur genau durch eine Variante realisiert ist. Aus diesen Mengen werden die Konfigurationsvarianten gebildet. Bei der „Billig“-Variante existiert folgende Menge von Konfigurationsvarianten: {{1, 12}, {1, 12, 13}, {1, 12, 14}, {1, 13}, {1, 13, 14}, {1, 14}}.
In der folgenden Kompositionsphase wird versucht, die Feature-Implementierungen aus jeder Konfigurationsvariante zu verschmelzen. Die einzige

gültige Lösung wird durch Verschmelzung der Feature-Implementierungen 1 und 14 erreicht. Bei allen anderen zeigt sich, dass bei gewählten Komponenten (`Hauptkoordinator` in 12 und `Fahrermoment` in 13) Ports nicht verbunden sind, deren Attribut `minLinks` ungleich 0 ist. Das ausgegebene Ergebnis ist in Abbildung 7.6 dargestellt.

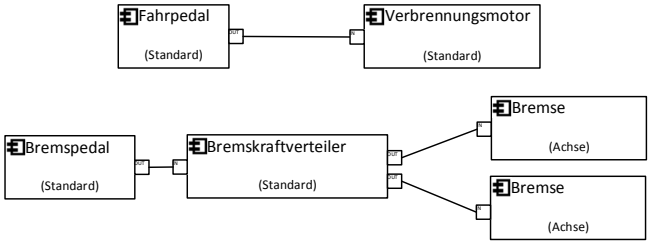


Abbildung 7.6: Die erstellte Produktarchitektur zur Feature-Konfiguration „Billig“

Anhand einer „Mittelklasse“-Konfiguration wird gezeigt, dass nach Vorgabe derselben Feature-Konfiguration auch mehrere unterschiedliche Produktarchitekturen vorgeschlagen werden können. Bei dieser Konfiguration werden die Features für die Integration der Geschwindigkeitsregelanlage (GRA) und der Abgasaufbereitung (SCR) ausgewählt. Diese beiden Features können jeweils durch alternative Feature-Implementierungen realisiert werden. Nach der Erstellung der Mengen von Konfigurationsvarianten, liegen mehrere Lösungen vor, die syntaktisch korrekt verschmolzen werden können. Drei dieser Lösungen werden im Folgenden gezeigt. In Abbildung 7.7 ist die Featurekonfiguration für diese drei ausgegebenen Produktarchitekturen dargestellt.

Vergleicht man die vorgeschlagenen Produktarchitekturen, die in Abbildung 7.8 und Abbildung 7.9 dargestellt sind, fällt auf, dass die Reihenfolge der Abgaskomponenten voneinander abweicht. Diese Abweichung ist ein Resultat, das direkt aus den technischen Randbedingungen des Kontextes der Domäne folgt. Wie in Abschnitt 3.1 beschrieben ist, gibt es mehrere Varianten von Abgassystemen, die verbaut sein können. Welche Produktarchitektur der Architekt auswählen wird, hängt davon ab, welche Hardware-Variante tatsächlich verbaut ist.

Bei einem Vergleich der Architekturen aus Abbildung 7.8 und Abbildung 7.10 ist eine andere Abweichung erkennbar. Während in Abbildung 7.8

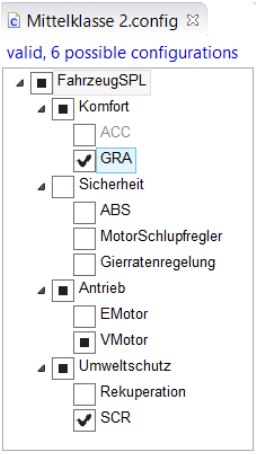


Abbildung 7.7: Diese Konfiguration einer Mittelklasse-Variante verursacht die Ableitung mehrerer Architekturen

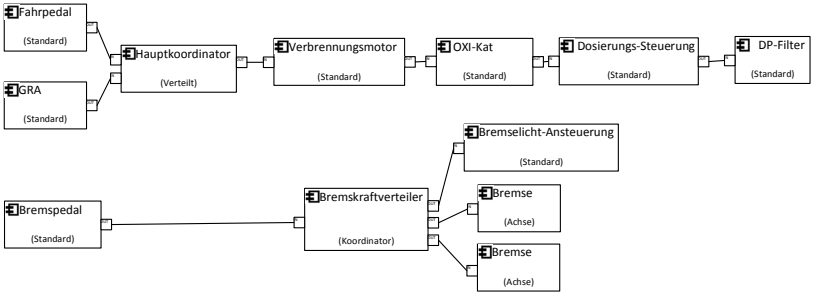


Abbildung 7.8: Diese Produktarchitektur repräsentiert eine valide Variante eines Mittelklasse-Fahrzeugs mit ACC und SCR

eine Beziehung zwischen `Hauptkoordinator` und `Bremskraftverteiler` fehlt, existiert diese in Abbildung 7.10.

Der Grund für diese Abweichung liegt in alternativen Feature-Implementierungen für Komfort-Features. In jedem Fall muss ein Koordinator zwei Momentanforderer (`Fahrpedal` und `GRA`) koordinieren. Die Beziehung zwischen `Hauptkoordinator` und `Bremskraftverteiler` ist aber nicht bei jedem Komfort-Feature notwendig. Bei der Auswahl der GRA darf

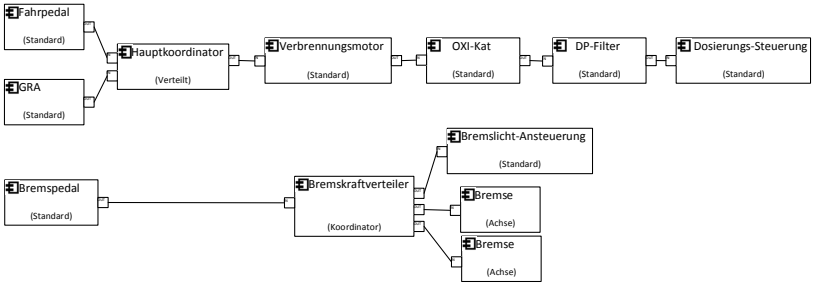


Abbildung 7.9: Auch diese Produktarchitektur ist eine valide „Mittelklasse“

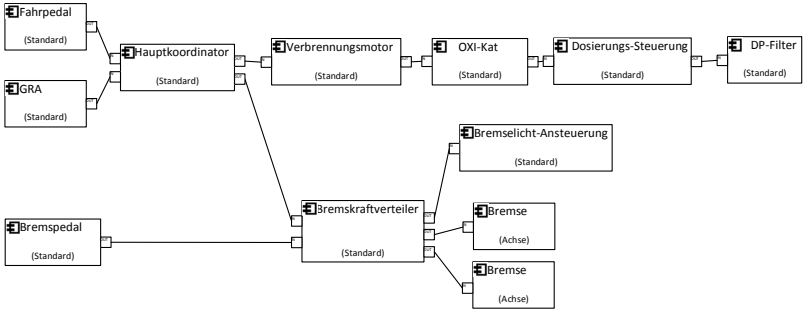


Abbildung 7.10: Diese Produktarchitektur ist nicht valide, da das GRA nicht bremsen darf

die Verbindung nicht existieren, da die Geschwindigkeitsregelanlage nicht bremsen kann beziehungsweise darf.

Ist für eine Konfigurationsvariante die Feature-Implementierung 5 für `GRA` und `Komfort` enthalten, wird die entsprechende Verbindung nicht vorgesehen. Allerdings kann auch Feature-Implementierung 5 für `GRA` und 20 für `Komfort` ausgewählt werden. Der Softwarearchitekt muss diese nicht gewünschte Verbindung erkennen und die Lösung als *nicht gültig* markieren.

Aus den grundsätzlich gültigen Architekturen wählt der Architekt die Variante aus, die sowohl die fachlichen als auch die technischen Randbedingungen erfüllt. Das Tool wird auf Basis dieser Architektur die an

die Architekturelemente geknüpften Informationen nutzen, um damit die Menge der entsprechenden Komponenten auszugeben. Zusätzlich wird ein Parametersatz ausgegeben, der sowohl die Konfigurationen der entsprechenden Komponenten festlegt als auch die Beziehungen zwischen den Komponenten definiert.

7.3 Zusammenfassung

In diesem Kapitel wurde an einem Beispiel demonstriert, dass das Konzept grundsätzlich anwendbar ist. Im ersten Teil wird gezeigt, welche Artefakte während des *domain engineering*s modelliert worden sind. Sechs verschiedene Strukturfragmente sind ausreichend, um die Beziehungsgeflechte aller Produkte zu beschreiben. Die Komponentenbibliothek enthält 18 Komponenten, wobei die meisten der Komponenten mit nur einer Komponentenkonfiguration auskommen. Die variantenreichste Komponente verfügt über sechs alternative Konfigurationen. Das Bindungsmodell enthält 21 Featureimplementierungen, die die Komponenten mit den Strukturfragmenten und den Features in Beziehung setzen.
Im zweiten Teil des Kapitels wird exemplarisch gezeigt, was im *application engineering* durchgeführt wird. Im ersten Beispiel sind die Ergebnisse der einzelnen Ableitungsphasen und die resultierende Produktarchitektur am Beispiel der Fahrzeugvariante „Billig“ beschrieben. Das zweite Beispiel basiert auf der Feature-Konfiguration einer „Mittelklasse“. Hier ist gezeigt, dass mehrere alternative Produktarchitekturen abgeleitet werden. Darunter befindet sich auch eine invalide Architektur, die der Architekt entsprechend markieren muss. Aus den verbleibenden wählt der Architekt die Variante, die auch die technischen Randbedingungen erfüllt.

Kapitel 8

Zusammenfassung und Ausblick

In diesem Kapitel werden die erarbeiteten Ergebnisse zusammengefasst. Das Ziel ist, die Frage zu beantworten, wie ein Architekt durch einen automatisierten Prozess bei der Erstellung von Produktarchitekturen im Umfeld einer feature-orientierten Softwareproduktlinie unterstützt werden kann.
Den Abschluss des Kapitels bildet ein Ausblick auf zukünftige Arbeiten auf Basis der in dieser Dissertation gewonnenen Erkenntnisse.

8.1 Zusammenfassung der Ergebnisse

Die Komplexität von Softwaresystemen nimmt ständig zu. Zum einen muss mehr Funktionalität integriert werden, zum anderen müssen die Systeme immer variabler werden. Sei es, weil die Anforderungen an die Funktionsauswahl von verschiedenen Nutzern stärker voneinander abweichen oder weil die Systeme in verschiedenen Umgebungen eingebettet werden. Dabei erschweren auch die Randbedingungen wie zunehmender Zeit- und Kostendruck die Entwicklung.
Das Konzept der Softwareproduktlinien adressiert diese Probleme. Durch Erstellung und Verknüpfung verschiedener Variabilitätsmodelle lässt sich die Komplexität in der Entwicklung reduzieren. Trotzdem ist die Modellierung von Softwarearchitekturen in softwareintensiven und variantenreichen Systemen weiterhin sehr schwierig. Die eingebetteten Systeme in Bereichen der Automobil-Industrie sind hierfür ein gutes Beispiel. Das Kernziel dieser Dissertation ist die Beantwortung der Forschungsfrage: „Wie kann man in einer feature-orientierten Softwareproduktlinie automatisiert komponentenbasierte Architekturen generieren?“
Es wird gezeigt, dass in einer Softwareproduktlinie die Variabilität auf unterschiedlichen Ebenen modelliert werden kann. Im *problem space* werden die rein fachlichen Aspekte mit Featuremodellen beschrieben. Die Softwareartefakte, aus denen die Produkte bestehen, bilden den *solution space*.

Auch auf dieser technischen Ebene muss Variabilität modelliert werden können. Durch den Einsatz modellgetriebener Entwicklungstechniken kann die Verwaltung dieser Variabilität vereinfacht werden (s. Kapitel 2). Der Fokus in dieser Dissertation liegt auf der Modellierung der Variabilität in der statischen Architektur komponentenbasierter Systeme.
An einem Beispiel, das angelehnt ist an eine Fahrzeug-Softwareproduktlinie, wird in Kapitel 3.1 gezeigt, worin sich verschiedene Produkte in einer Produktlinie unterscheiden. Die Untersuchung dieser Unterschiede zeigt, dass es drei Erscheinungsformen von Variabilität in der Software-Produktlinienarchitektur gibt.
Die erste identifizierte Form ist die Komponentenvariabilität. Bei dieser Form unterscheiden sich die abgeleiteten Architekturen zweier Produkte darin, dass an einer bestimmten Stelle unterschiedliche Komponenten zum Einsatz kommen. Die Konfigurationsvariabilität ist die zweite Variabilitätsform. Hierbei liegt die Variabilität komplett im Inneren einer Komponente. Die Wahl einer bestimmten Variante legt eine Menge von außen sichtbarer Eigenschaften der Komponente fest. Die dritte Form ist die Systemstrukturvariabilität. Diese Form bewirkt, dass sich die Unterschiede in Produktarchitekturen im Beziehungsgeflecht zeigen, das die Komponenten im jeweiligen System untereinander eingehen.
An dem Beispiel der Fahrzeugproduktlinie wurde des Weiteren untersucht, wo die Ursachen für die Notwendigkeit der vorhandenen Variabilität liegen. Zum einen ist die Variabilität in der Softwarearchitektur in der Variabilität des *problem space* begründet. Produkte, die unterschiedliche Features realisieren, müssen mindestens eine der Variabilitätsformen aufweisen. Eine weitere Ursache sind verschiedene technische Aspekte, die nicht im Featuremodell erfasst sind.
Diese Beobachtungen – die Variabiabilitätsformen und die Ursachen für Variabilität – bilden zusätzliche Randbedingungen, die bei der Beantwortung der Forschungsfrage berücksichtigt werden müssen. Die Lösung für dieses Problem wird durch das Erreichen von drei Teilzielen definiert.
Das erste Teilziel lautet: „Es muss ein Entwicklungsprozess beschrieben werden, der die Schritte definiert, die für die Erstellung einer Softwareproduktlinie notwendig sind. Der Schwerpunkt liegt dabei auf der Architekturentwicklung." Dieses Ziel wurde durch die Beschreibung des Prozesses in Kapitel 4.1 erreicht. Die Unterteilung in die zwei Hauptschritte *domain engineering* und *application engineering* basiert dabei auf etablierten Prozessen. Bei der Verfeinerung dieser Schritte werden die oben genannten Randbedingungen berücksichtigt. Für jede Variabilitätsform ist ein eigener Modellierungsschritt vorgesehen. Die Abhängigkeit einzelner Varianten von fachlichen Aspekten wird im Prozessschritt der Kopplung von *problem*

space und *solution space* modelliert. Durch technische Aspekte bedingte Variabilität wird im *application engineering* durch den Architekten aufgelöst. Er muss aus verschiedenen Lösungen die passende Produktarchitektur auswählen.

Das zweite Teilziel der Dissertation lautet: „Es soll ein formales Modellierungskonzept definiert werden, mit der sich die einzelnen Teile und Zusammenhänge modellieren lassen, sodass eine automatisierte Architekturableitung ermöglicht wird." In Kapitel 4.2 wird ein entsprechendes Konzept vorgestellt. Es können verschiedene Modellelemente erstellt werden, die jeweils eine Variante für genau eine Variabilitätsform repräsentieren. Ein Strukturfragment beschreibt einen Ausschnitt eines Beziehungsgeflechts. Systemstrukturvariabilität wird durch die Verschmelzung unterschiedlicher Strukturfragmente erreicht. Eine Komponentenkonfiguration repräsentiert genau eine Kombination von Eigenschaften, die eine Komponente in einem Produkt aufweisen kann. Die Zuordnung mehrerer Komponentenkonfigurationen zu einer Komponente realisiert die Konfigurationsvariabilität. Die Möglichkeit mehrere Komponenten an ein Strukturfragment zu binden, erzeugt die notwendige Komponentenvariabilität.

Durch die Bindung einzelner Varianten an Features wird der Einfluss des *problem space* auf die Variabilität im *solution space* berücksichtigt. Das Bindungsmodell sieht vor, dass mehrere alternative Produktarchitekturen nach Vorgabe einer Featurekonfiguration abgeleitet werden können. Diese Variabilität folgt aus der Berücksichtigung alternativer technischer Randbedingungen.

Die in Kapitel 5 eingeführte Modellierungssprache repräsentiert die Formalisierung des Konzepts. Das zugrundeliegende Metamodell ist so gestaltet, dass sowohl die Modellierung der einzelnen Artefakte im definierten Prozess sehr gut umgesetzt werden kann als auch die Ableitung einzelner Produktarchitekturen.

Das letzte Teilziel lautet: „Anhand eines Werkzeugprototypen soll die Anwendbarkeit des Konzepts mit Hilfe des Beispiels demonstriert werden." In Kapitel 6 ist der Aufbau eines Prototypen beschrieben. Mit der entwickelten Software ist es möglich, die Artefakte der Softwareproduktlinie aus Kapitel 3.1 zu erstellen (s. Kapitel 7). Aus dem Softwareproduktlinienmodell lassen sich die gewünschten Produktarchitekturen ableiten.

Das Erreichen dieser drei Teilziele bedeutet, dass auch das durch die Forschungsfrage definierte Kernziel in dieser Dissertation erreicht ist. Durch den Einsatz des fragmentbasierten Modellierungskonzepts lassen sich automatisiert komponentenbasierte Architekturen in einer feature-orientierten Softwareproduktlinie generieren.

8.2 Ausblick

Es gibt mehrere vielversprechende Ideen, das Konzept zu erweitern, um es noch besser auch in anderen Domänen einsetzen zu können. Drei besonders interessante sollen am Ende dieser Dissertation vorgestellt werden.
Der erste Ansatz ist die Erweiterung des Konzepts um die Möglichkeit einer hierarchischen Modellierung. Dafür geht man davon aus, dass eine Komponente ein logisches Konstrukt ist, das eine Komposition weiterer Komponenten repräsentiert. Verfügt diese Komponente über mehrere Konfigurationen, wird auch die Komposition im Inneren einer Variabilität unterliegen. Es ist zu erwarten, dass hierbei die gleichen Variabilitätsformen auftreten, wie sie auf der in dieser Dissertation betrachteten Systemebene identifiziert wurden. Es liegt daher nah, dass sich der fragmentbasierte Modellierungsansatz auch für die Beschreibung der Variabilität dieser Komposition eignet.
Eine wesentliche Herausforderung stellt die Erweiterung des Bindungsmodells dar. Eine interessante Aufgabe ist dabei die Untersuchung der Auswirkungen der Variabilitätsinformation auf verschiedenen Hierarchieebenen.
Eine andere mögliche Fortsetzung der Arbeit könnte sich mit alternativen Möglichkeiten zur Verknüpfung mehrerer Strukturfragmente zu einem größeren Beziehungsgeflecht befassen. In der vorgestellten Domäne zeigt sich, dass eine Komponente nicht immer eindeutig für die Realisierung eines Features zuständig ist. Vielmehr ist eine Komponente oft an verschiedenen Kompositionen beteiligt, die jeweils unterschiedliche Features realisieren. Da Strukturfragmente das Beziehungsgeflecht der Kompositionen beschreiben, müssen sie an der Stelle der entsprechenden Komponente überlappen können. In dieser Dissertation wurde die Überlappungsmöglichkeit von Strukturfragmenten gezielt ausgenutzt, um das Beziehungsgeflecht eines Produkts durch Verschmelzung mehrerer Strukturfragmente zu definieren. In anderen Domänen wird die Anzahl der Überlappungen deutlich geringer ausfallen. Um auch in solchen Fällen effizient mit Strukturfragmenten arbeiten zu können, wäre es sehr hilfreich, auch andere Techniken zur Kombination von Strukturfragmenten vorzusehen. Dafür ist einerseits zu untersuchen, wie Strukturfragmente durch diese zusätzliche Anforderung erweitert werden müssen. Andererseits muss auch in diesem Fall das Bindungsmodell erweitert werden.
In dieser Dissertation werden zwei Ursachen für die Notwendigkeit von Variabilität in den Softwareartefakten identifiziert. Zum einen sind das unterschiedliche fachliche Aspekte, zum anderen unterschiedliche technische Aspekte in der Domäne. Der fragmentbasierte Ansatz berücksichtigt zwar

grundsätzlich beide Ursachen, für die Lösungsfindung einer automatisierten Ableitung von Architekturen lag der Fokus aber auf der Beziehung zu fachlichen Aspekten. Am Ende des Entwicklungsprozesses muss der Architekt aus verschiedenen Produktarchitekturen manuell eine geeignete auswählen.
Die Erweiterung des Fokus auf die Untersuchung der technischen Aspekte stellt eine weitere Möglichkeit zu Forschungen auf Basis dieser Arbeit dar. Es muss geklärt werden, welche Arten von technischen Randbedingungen auftreten können. Dabei ist die Frage interessant: Wie kann die Variabilität in diesen Arten beschrieben werden? Die Folge könnte einerseits die Anpassung des Bindungsmodells sein. Andererseits muss gegebenenfalls auch der Prozess erweitert werden.
Das in dieser Dissertation vorgestellte Konzept *Fragmentbasierte Softwarearchitekturen für Produktlinien* repräsentiert also nicht nur die Lösung auf die gestellte Forschungsfrage, sondern stellt darüber hinaus eine solide Grundlage für weitere Forschungsarbeiten dar.

Literatur

[14a] *GMF Tooling. Model Driven Architecture approach to domain of graphical editors*. The Eclipse Foundation. 2014. URL: http://www.eclipse.org/gmf-tooling/ (besucht am 23. 03. 2016).

[14b] *Graphical Modeling Framework*. The Eclipse Foundation. 2014. URL: http://wiki.eclipse.org/Graphical_Modeling_Framework/Documentation (besucht am 23. 03. 2016).

[16a] *ASCET Software-Produkte*. ETAS GmbH. 18. Apr. 2016. URL: http://www.etas.com/de/products/ascet_software_products.php (besucht am 18. 04. 2016).

[16b] *Community Software Architecture Definitions*. Software Engineering Institute - Carnegie Mellon University. 2016. URL: http://www.sei.cmu.edu/architecture/start/glossary/community.cfm#definitions (besucht am 02. 05. 2016).

[16c] *Eclipse Modeling Framework (EMF)*. The Eclipse Foundation. 2016. URL: http://www.eclipse.org/modeling/emf/ (besucht am 23. 03. 2016).

[16d] *Eclipse Platform*. The Eclipse Foundation. 2016. URL: https://projects.eclipse.org/projects/eclipse.platform (besucht am 16. 03. 2016).

[16e] *Eclipse4/RCP/Architectural Overview*. The Eclipse Foundation. 2016. URL: https://wiki.eclipse.org/Eclipse4/RCP/Architectural_Overview (besucht am 23. 03. 2016).

[16f] *Equinox OSGi*. The Eclipse Foundation. 2016. URL: http://www.eclipse.org/equinox/ (besucht am 23. 03. 2016).

[16g] *Graphical Modeling Project (GMP)*. The Eclipse Foundation. 2016. URL: http://www.eclipse.org/modeling/gmp/ (besucht am 13. 03. 2016).

[AJB12] Arne Haber, Jan Oliver Ringert und Bernhard Rumpe. *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems: Technical Report*. Hrsg. von RWTH Aachen. 2012. URL: http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2012/2012-03.pdf (besucht am 13. 10. 2016).

[AK09] Sven Apel und Christian Kästner. "An Overview of Feature-Oriented Software Development." In: *Journal of Object Technology* 8.5 (2009), S. 49–84.

[Aut16] Verband der Automobilindustrie e.V. *AdBlue*. VDA. 19. Juni 2016. URL: https://www.vda.de/dam/vda/publications/1380205585_de_469110321.pdf (besucht am 24. 11. 2015).

[BCK98] Len Bass, Paul Clements und Rick Kazman. *Software Architecture in Practice*. 1. Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201199300.

[CA05] Krzysztof Czarnecki und Michał Antkiewicz. "Mapping features to models: A template approach based on superimposed variants". In: *International conference on generative programming and component engineering*. Springer. 2005, S. 422–437.

[CN01] Paul Clements und Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001. ISBN: 978-0-201-70332-0.

[Con10] World Wide Web Consortium. *XML Path Language (XPath) 2.0 (second edition)*. 14. Dez. 2010. URL: http://www.w3.org/TR/xpath20/ (besucht am 13. 10. 2016).

[CW02] Tony Clark und Jos Warmer. *Object Modeling with the OCL. The Rationale behind the Object Constraint Language*. Lecture Notes in Computer Science. Berlin: Springer-Verlag, 2002. ISBN: 3-540-43169-1. DOI: 10.1007/3-540-45669-4.

[Eur07] Europäisches Parlament und Rat der europäischen Union. *Verordnung (EG) Nr. 715/2007 des europäischen Parlaments und des Rates über die Typgenehmigung von Kraftfahrzeugen hinsichtlich der Emissionen von leichten Personenkraftwagen und Nutzfahrzeugen (Euro 5 und Euro 6) und über den Zugang zu Reparatur- und Wartungsinformationen für Fahrzeuge: EURO 5 und EURO 6*. 20. Juni 2007. URL: http://eur-lex.europa.eu/legal-content/DE/TXT/PDF/?uri=CELEX:32007R0715 (besucht am 17. 04. 2015).

[Fan11] Julie Street Fant. “Building domain specific software architectures from software architectural design patterns”. In: *2011 33rd International Conference on Software Engineering (ICSE)*. Mai 2011, S. 1152–1154. DOI: 10.1145/1985793.1986026.

[FGP13] Julie Street Fant, Hassan Gomaa und Robert G. Pettit. “A Pattern-Based Modeling Approach for Software Product Line Engineering”. In: *System Sciences (HICSS), 2013 46th Hawaii International Conference on*. Jan. 2013, S. 4985–4994. DOI: 10.1109/HICSS.2013.52.

[Gal+11] Matthias Galster u. a. “Variability in Software Architecture: Current Practice and Challenges”. In: *SIGSOFT Softw. Eng. Notes* 36.5 (Sep. 2011), S. 30–32. ISSN: 0163-5948. DOI: 10.1145/2020976.2020978. URL: http://doi.acm.org/10.1145/2020976.2020978.

[Gam+04] Erich Gamma u. a. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley Verlag, 2004. ISBN: 978-3-8273-2199-2.

[GBS01] Jilles van Gurp, Jan Bosch und Mikael Svahnberg. “On the Notion of Variability in Software Product Lines”. In: *In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*. IEEE Computer Society, 2001, S. 45–54.

[Gha+13] Mahboura Gharbi u. a. *Basiswissen für Softwarearchitekten*. dpunkt.verlag GmbH, 2013. ISBN: 9783898647915.

[Gmb07] ETAS GmbH. *Ascet V5.2 Benutzerhandbuch*. Stuttgart, 2007.

[GS09] Wolfgang Goerigk und Thomas Stahl. “Modellgetriebenes Softwareengineering - Der Beginn industrieller Softwareproduktion?” In: *Software Engineering 2009 - Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik 02.-06.03.2009 in Kaiserslautern*. 2009, S. 115–118. URL: http://subs.emis.de/LNI/Proceedings/Proceedings150/article4853.html.

[Hab+11a] Arne Haber u. a. “Delta Modeling for Software Architectures”. In: *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*. 2011, S. 1–10. URL: http://www.se-rwth.de/publications/AH.HR.BR.IS.DeltaMontiArc.MBEES2011.pdf (besucht am 13. 10. 2016).

[Hab+11b] Arne Haber u. a. “Delta-oriented Architectural Variability Using MontiCore”. In: *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. ECSA ’11. New York, NY, USA: ACM, 2011, 6:1–6:10. ISBN: 978-1-4503-0618-8. DOI: 10.1145/2031759.2031767. URL: http://doi.acm.org/10.1145/2031759.2031767.

[Hab+11c] Arne Haber u. a. “Hierarchical Variability Modeling for Software Architectures”. In: *2011 15th International Software Product Line Conference*. IEEE, 2011, S. 150–159. ISBN: 978-1-4577-1029-2.

[Her11] Sebastian Herold. “Architectural Compliance in Component-Based Systems”. Diss. Clausthal University of Technology, 2011. ISBN: 978-3-8439-0109-3. URL: http://www.dr.hut-verlag.de/978-3-8439-0109-3.html (besucht am 13. 10. 2016).

[Hil12] Martin Hillenbrand. “Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen: Dissertation”. Diss. Karlsruher Institut für Technologie, 2012. ISBN: 978-3-86644-803-2.

[HS12] Peter Hruschka und Gernot Starke. *Das arc42 Template*. 2012. URL: http://arc42.de/template/index.html (besucht am 15. 05. 2016).

[Inc15] BigLever Software Inc. *BigLever Software Gears*. 2015. URL: http://www.biglever.com (besucht am 24. 11. 2015).

[Kan+90] Kyo C Kang u. a. *Feature-oriented domain analysis (FODA) feasibility study*. Techn. Ber. DTIC Document, 1990.

[Kan+98] Kyo C Kang u. a. “FORM: A feature-oriented reuse method with domain-specific reference architectures”. In: *Annals of Software Engineering* 5.1 (1998), S. 143–168.

[KHR14] Marco Körner, Sebastian Herold und Andreas Rausch. “Composition of Applications Based on Software Product Lines Using Architecture Fragments and Component Sets”. In: *Proceedings of the WICSA 2014 Companion Volume*. WICSA ’14 Companion. Sydney, Australia: ACM, 2014, 13:1–13:4. ISBN: 978-1-4503-2523-3. DOI: 10.1145/2578128.2578239. URL: http://doi.acm.org/10.1145/2578128.2578239.

[Kle08] Anneke Kleppe. *Software Language Engineering. Creating Domain-Specific Languages Using Metamodels*. Addison Wesley, 2008. ISBN: 978-0321553454.

[Kru95] P. B. Kruchten. "The 4+1 View Model of architecture". In: *IEEE Software* 12.6 (Nov. 1995), S. 42–50. ISSN: 0740-7459. DOI: 10.1109/52.469759.

[Küb09] Sebastian Kübeck. *Software-Sanierung. Weiterentwicklung, Testen und Refactoring bestehender Software*. mitp-Verlag, 2009. ISBN: 9783826650727.

[KW04] Anneke Kleppe und Jos Warmer. *Object Constraint Language 2.0*. mitp-Verlag, 2004. ISBN: 3826614453.

[Lie+10] Jörg Liebig u. a. "An Analysis of the Variability in Forty Preprocessor-based Software Product Lines". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: ACM, 2010, S. 105–114. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806819. URL: http://doi.acm.org/10.1145/1806799.1806819.

[Loh+06] Daniel Lohmann u. a. "A Quantitative Analysis of Aspects in the eCos Kernel". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. EuroSys '06. Leuven, Belgium: ACM, 2006, S. 191–204. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217954. URL: http://doi.acm.org/10.1145/1217935.1217954.

[LWE11] Duc Le, E. Walkingshaw und M. Erwig. "#ifdef confirmed harmful: Promoting understandable software variation". In: *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*. Sep. 2011, S. 143–150. DOI: 10.1109/VLHCC.2011.6070391.

[Lyt+13] Ioanna Lytra u. a. "On the Interdependence and Integration of Variability and Architectural Decisions". In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '14. Sophia Antipolis, France: ACM, 2013, 19:1–19:8. ISBN: 978-1-4503-2556-1. DOI: 10.1145/2556624.2556634. URL: http://doi.acm.org/10.1145/2556624.2556634.

[Nie+07] Dirk Niebuhr u. a. "DAiSI-Dynamic Adaptive System Infrastructure". In: *Fraunhofer Institut Experimentelles Software Engineering, IESE-Report No* 51 (2007).

[Obj12a] Object Management Group. *Object Constraint Language (OCL): formal/2012-05-09*. 2012. URL: http://www.omg.org/spec/OCL/ISO/19507/PDF/ (besucht am 13. 10. 2016).

[Obj12b] Object Management Group. *Unified Modeling Language (OMG UML): Infrastructure: formal/2012-05-06*. 2012. URL: http://www.omg.org/spec/UML/ISO/19505-1/PDF/ (besucht am 13. 10. 2016).

[Obj12c] Object Management Group. *Unified Modeling Language (OMG UML): Superstructure: formal/2012-05-07*. 2012. URL: http://www.omg.org/spec/UML/ISO/19505-2/PDF/ (besucht am 13. 10. 2016).

[Obj14] Object Management Group. *Model Driven Architecture (MDA): The MDA Guide Rev. 2.0*. Object Management Group. 2014. URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01 (besucht am 09. 05. 2016).

[OMG10] OMG. *The MDA Foundation Model*. Object Management Group. 2010. URL: http://www.omg.org/cgi-bin/doc?ormsc/10-09-06 (besucht am 09. 05. 2016).

[OMG15] OMG. *Meta Object Facility (MOF) Core Specification*. Object Management Group. Juni 2015. URL: http://www.omg.org/spec/MOF/2.5/PDF (besucht am 09. 05. 2016).

[Omm+00] R. van Ommering u. a. "The Koala component model for consumer electronics software". In: *Computer* 33.3 (März 2000), S. 78–85. ISSN: 0018-9162. DOI: 10.1109/2.825699.

[Par72] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Commun. ACM* 15.12 (Dez. 1972), S. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623. URL: http://doi.acm.org/10.1145/361598.361623.

[Pre+07] Alexander Pretschner u. a. "Software engineering for automotive systems: A roadmap". In: *2007 Future of Software Engineering*. IEEE Computer Society. 2007, S. 55–71.

[pur15] pure-systems. *pure::variants*. 24. Nov. 2015. URL: http://www.pure-systems.com (besucht am 24. 11. 2015).

[Rob15] Robert Bosch GmbH. *Denoxtronic 5 – Dosiersystem für AdBlue in SCR-Systemen*. 24.05.2015. URL: http://produkte.bosch-mobility-solutions.de/media/db_application/downloads/pdf/antrieb/de_5/DS_Sheet_Denoxtronic5-Dosiersystem_20120720.pdf (besucht am 24. 04. 2015).

[SBD11] Ina Schaefer, Lorenzo Bettini und Ferruccio Damiani. "Compositional Type-Checking for Delta-oriented Programming". In: *Proceedings of the tenth international conference on Aspect-oriented software development, AOSD '11*. ACM, 2011, S. 43–56. URL: http://doi.acm.org/10.1145/1960275.1960283.

[SC92] Henry Spencer und Geoff Collyer. "# ifdef considered harmful, or portability experience with C News". In: *USENIX Summer 1992 Technical Conference*. USENIX The Advanced Computing Systems Association, 1992. URL: https://www.usenix.org/legacy/publications/library/proceedings/sa92/spencer.pdf (besucht am 13. 10. 2016).

[Sch+12] Ina Schaefer u. a. "Software diversity: state of the art and perspectives". In: *International Journal on Software Tools for Technology Transfer* 14.5 (2012), S. 477–495. ISSN: 1433-2779. DOI: 10.1007/s10009-012-0253-y.

[SD07] Marco Sinnema und Sybren Deelstra. *Classifying variability modeling techniques*. 2007. DOI: 10.1016/j.infsof.2006.08.001. URL: http://www.msinnema.nl/sinnema_deelstra_classifying_variability_modeling_techniques.pdf (besucht am 06. 12. 2012).

[SGG12] Julie Street Fant, Hassan Gomaa und Robert G. Pettit IV. *Software Product Line Engineering of Space Flight Software*. Hrsg. von IEEE. 2012. URL: http://ieeexplore.ieee.org/ielx5/6220311/6229758/06229769.pdf?tp=&arnumber=6229769&isnumber=6229758 (besucht am 16. 05. 2013).

[SH09] Gernot Starke und Peter Hruschka. *Software-Architektur kompakt. angemessen und zielorientiert*. Heidelberg: Spektrum Akademischer Verlag, 2009. ISBN: 978-3-8274-2093-0.

[SRG11] Klaus Schmid, Rick Rabiser und Paul Grünbacher. "A Comparison of Decision Modeling Approaches in Product Lines". In: *Proceedings of the 5th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS'11)*. Hrsg. von Patrick Heymans, Krzysztof Czarnecki und Ul-

rich W. Eisenecker. ACM, 2011, S. 119–126. DOI: 10.1145/1944892.1944907.

[Sta+07] Thomas Stahl u. a. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt. verlag, 2007. ISBN: 9783898644488.

[Sta08] Gernot Starke. *Effektive Software-Architekturen*. Carl Hanser Verlag GmbH & Co. KG, 2008. ISBN: 9783446412156.

[Ste+08] Dave Steinberg u. a. *EMF: Eclipse Modeling Framework*. 2nd Edition. Eclipse Series. 2008. ISBN: 978-0-321-33188-5.

[SVB05] Mikael Svahnberg, Jilles Van Gurp und Jan Bosch. "A taxonomy of variability realization techniques". In: *Software: Practice and Experience* 35.8 (2005), S. 705–754.

[Szy98] Clemens Szyperski. *Component software: beyond object-oriented programming*. Addison Wesley Longman Limited, 1998. ISBN: 0201178885.

[Thü+14] Thomas Thüm u. a. "FeatureIDE: An Extensible Framework for Feature-oriented Software Development". In: *Sci. Comput. Program.* 79 (Jan. 2014), S. 70–85. ISSN: 0167-6423. DOI: 10.1016/j.scico.2012.06.002. URL: http://dx.doi.org/10.1016/j.scico.2012.06.002.

[Tit03] Peter Tittmann. *Graphentheorie. Eine anwendungsorientierte Einführung*. Leipzig, Germany: Fachbuchverlag Leipzig im Carl Hanser Verlag, 2003. ISBN: 3446223436.

[TM15] Thomas Thüm und Jens Meinicke. *FeatureIDE: Background. Short overview of Feature-Oriented Software Development*. 4. März 2015. URL: http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/slides/featureide-0-background.pdf (besucht am 16. 03. 2016).

[Unb08] H. Unbehauen. *Regelungstechnik I: Klassische Verfahren zur Analyse und Synthese linearer kontinuierlicher Regelsysteme, Fuzzy-Regelsysteme*. Studium Technik. Vieweg+Teubner Verlag, 2008. ISBN: 9783834804976.

[VG07] M. Voelter und I. Groher. "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development". In: *Software Product Line Conference, 2007. SPLC 2007. 11th International*. 2007, S. 233–242. DOI: 10.1109/SPLINE.2007.23.

[Vog15] Lars Vogel. *Eclipse Rich Client Platform. The complete guide to Eclipse application development*. Mit einem Vorw. von Mike Milinkovich. vogella series. Lars Vogel, 13. Mai 2015. ISBN: 978-3943747133.

[Vog16] Lars Vogel. *The Architecture of Eclipse*. 2016. URL: `http://www.vogella.com/tutorials/EclipseRCP/article.html#architecture` (besucht am 13. 10. 2016).

[Wei15] G. Weiß. *Design of self-adaptation in distributed embedded systems*. München: Verlag Dr. Hut, 2015. ISBN: 978-3-8439-1966-1.